DB2 Universal Database for OS/390 and z/OS

IBM

Application Programming and SQL Guide

Version 7

DB2 Universal Database for OS/390 and z/OS

IBM

Application Programming and SQL Guide

Version 7

> **Note**
>
> Before using this information and the product it supports, be sure to read the general information under "Notices" on page 937.

Contents

Part 6. Additional programming techniques 487

About this book

This book discusses how to design and write application programs that access DB2® for OS/390® (DB2), a highly flexible relational database management system (DBMS).

Important

In this version of DB2 for OS/390 and z/OS, some utility functions are available as optional products. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

Who should read this book

This book is for DB2 application developers who are familiar with Structured Query Language (SQL) and who know one or more programming languages that DB2 supports.

Product terminology and citations

In this book, DB2 Universal Database™ Server for OS/390 and z/OS is referred to as "DB2 for OS/390 and z/OS." In cases where the context makes the meaning clear, DB2 for OS/390 and z/OS is referred to as "DB2." When this book refers to other books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM® DATABASE 2™ Universal Database Server for OS/390 and z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 for OS/390 and z/OS, this book uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

DB2 Represents either the DB2 licensed program or a particular DB2 subsystem.

C and C language
 Represent the C programming language.

CICS® Represents CICS/ESA® and CICS Transaction Server for OS/390.

IMS™ Represents IMS or IMS/ESA®.

MVS Represents the MVS element of OS/390.

OS/390
 Represents the OS/390 or z/OS operating system.

RACF®
 Represents the functions that are provided by the RACF component of the SecureWay® Security Server for OS/390 or by the RACF component of the OS/390 Security Server.

How to read the syntax diagrams

The following rules apply to the syntax diagrams used in this book:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a statement.

  The ──► symbol indicates that the statement syntax is continued on the next line.

  The ►── symbol indicates that a statement is continued from the previous line.

  The ──►◄ symbol indicates the end of a statement.

- Required items appear on the horizontal line (the main path).

  ►►──*required_item*──►◄
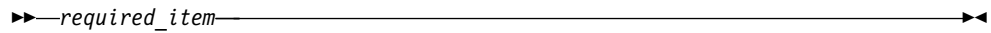
- Optional items appear below the main path.

  ►►──*required_item*──►◄
         └─*optional_item*─┘
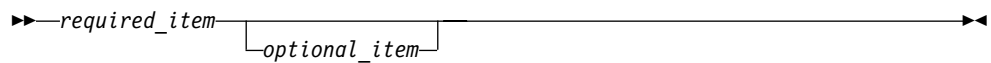
  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

         ┌─*optional_item*─┐
  ►►──*required_item*──┴──────────────┴───────────────────────►◄
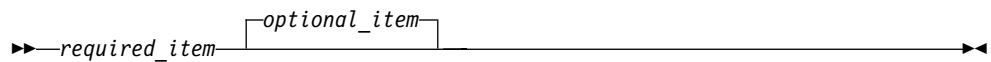
- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

  ►►──*required_item*──┬─*required_choice1*─┬───────────────────►◄
                └─*required_choice2*─┘
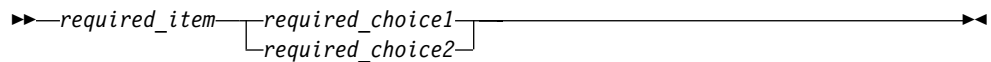
  If choosing one of the items is optional, the entire stack appears below the main path.

  ►►──*required_item*───►◄
          ├─*optional_choice1*─┤
          └─*optional_choice2*─┘
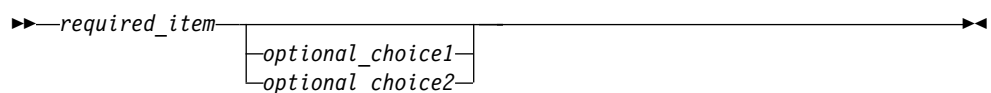
  If one of the items is the default, it appears above the main path and the remaining choices are shown below.

         ┌─*default_choice*─┐
  ►►──*required_item*──┼──────────────────┼──────────────────────►◄
         ├─*optional_choice*─┤
         └─*optional_choice*─┘
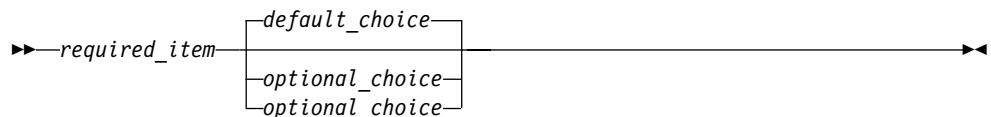
- An arrow returning to the left, above the main line, indicates an item that can be repeated.

```
                    ┌─────────────────────┐
                    │                     │
▶▶──required_item──┴──repeatable_item──┘──────────────────────────────▶◀
```
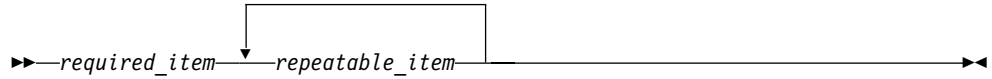
If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                    ┌──────,──────────────┐
                    │                     │
▶▶──required_item──┴──repeatable_item──┘──────────────────────────────▶◀
```
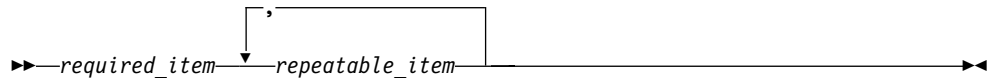
A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 for OS/390 and z/OS documentation. You can use any of the following methods to provide comments:

- Send your comments by e-mail to db2pubs@vnet.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title, page number, or a help topic title).
- Send your comments from the Web. Visit the Web site at:

  http://www.ibm.com/software/db2os390

  The Web site has a feedback page that you can use to send comments.
- Complete the readers' comment form at the back of the book and return it by mail, by fax (800-426-7773 for the United States and Canada), or by giving it to an IBM representative.

Summary of changes to this book

The principal changes to this book are:

- Information that is duplicated in other books has been removed from "Chapter 1. Retrieving data" on page 3.

- Chapter 7. Using a cursor to retrieve a set of rows explains how to use scrollable cursors.

- Chapter 24. Using stored procedures for client/server processing contains information on enhancements to stored procedures, such as the ability to issue COMMIT and ROLLBACK statements in stored procedures.

- Appendix I. Stored procedures shipped with DB2 is a new appendix that describes the WLM environment refresh stored procedure.

xxi

Part 1. Using SQL queries

Chapter 1. Retrieving data

You can retrieve data using the SQL statement SELECT to specify a result table. This chapter describes how to use SELECT statements interactively to retrieve data from DB2 tables.

For more advanced topics on using SELECT statements, see "Chapter 4. Using subqueries" on page 43, "Chapter 19. Planning to access distributed data" on page 367, and Chapter 4 of *DB2 SQL Reference*.

Examples of SQL statements illustrate the concepts that this chapter discusses. Consider developing SQL statements similar to these examples and then execute them dynamically using SPUFI or Query Management Facility (QMF).

Result tables

The data retrieved through SQL is always in the form of a table, which is called a *result table*. Like the tables from which you retrieve the data, a result table has rows and columns. A program fetches this data one row at a time.

Example: SELECT statement: This SELECT statement retrieves the last name, first name, and phone number of employees in department D11 from the sample employee table:

```
SELECT LASTNAME, FIRSTNME, PHONENO
  FROM DSN8710.EMP
  WHERE WORKDEPT = 'D11'
  ORDER BY LASTNAME;
```

The result table looks like this:

```
LASTNAME          FIRSTNME        PHONENO
===============   ============    ========
ADAMSON           BRUCE           4510
BROWN             DAVID           4501
JOHN              REBA            0672
JONES             WILLIAM         0942
LUTZ              JENNIFER        0672
PIANKA            ELIZABETH       3782
SCOUTTEN          MARILYN         1682
STERN             IRVING          6423
WALKER            JAMES           2986
YAMAMOTO          KIYOSHI         2890
YOSHIMURA         MASATOSHI       2890
```

The result table displays in this form after SPUFI fetches and formats it. The format of your results might be different.

Data types

When you create a DB2 table, you define each column to have a specific data type. The data type can be a built-in data type or a distinct type. This section discusses built-in data types. For information on distinct types, see "Chapter 15. Creating and using distinct types" on page 299. The data type of a column determines what you can and cannot do with it. When you perform operations on columns, the data must be compatible with the data type of the referenced column. For example, you cannot insert character data, like a last name, into a column whose data type is numeric. Similarly, you cannot compare columns containing incompatible data types.

3

To better understand the concepts presented in this chapter, you must know the data types of the columns to which an example refers. As shown in Figure 1, the data types have four general categories: string, datetime, numeric, and ROWID.
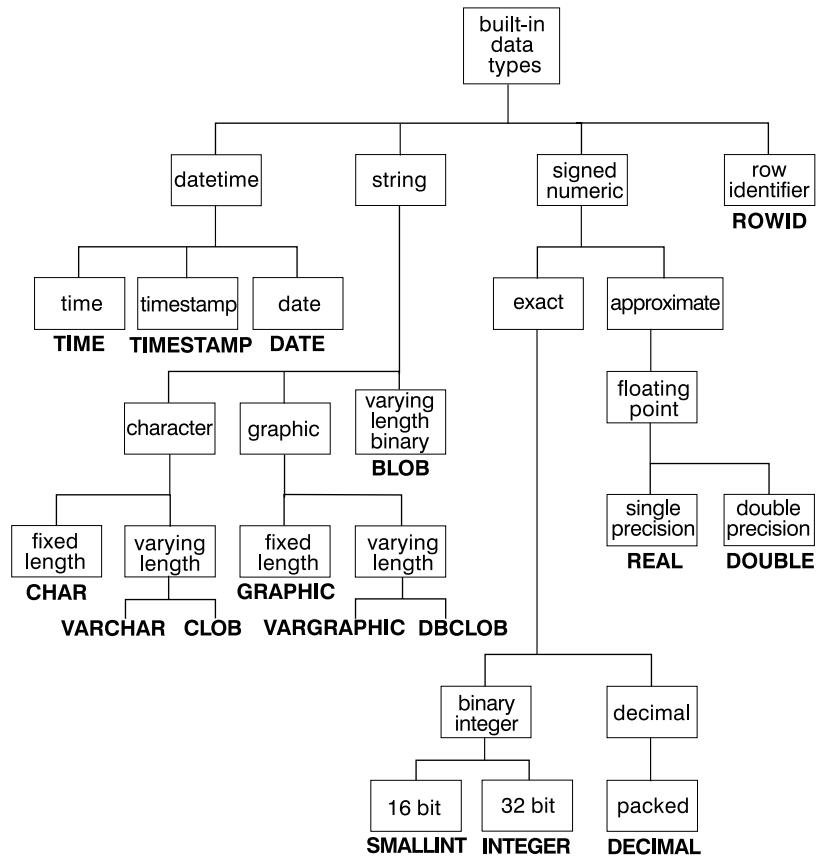
Figure 1. DB2 data types

For more detailed information on each data type, see Chapter 2 of *DB2 SQL Reference*.

Table 1 on page 5 shows whether operands of any two data types are compatible (Yes) or incompatible (No).

Table 1. Compatibility of data types for assignments and comparisons. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operands	Binary integer	Decimal number	Floating point	Char-acter string	Graphic string	Binary string	Date	Time	Time-stamp	Row ID	Distinct type
Binary Integer	Y	Y	Y	N	N	N	N	N	N	N	2
Decimal Number	Y	Y	Y	N	N	N	N	N	N	N	2
Floating Point	Y	Y	Y	N	N	N	N	N	N	N	2
Character String	N	N	N	Y	N[4,5]	N[3]	1	1	1	N	2
Graphic String	N	N	N	N[4,5]	Y	N	1,4	1,4	1,4	N	2
Binary String	N	N	N	N[3]	N	Y	N	N	N	N	2
Date	N	N	N	1	1,4	N	Y	N	N	N	2
Time	N	N	N	1	1,4	N	N	Y	N	N	2
Time-stamp	N	N	N	1	1,4	N	N	N	Y	N	2
Row ID	N	N	N	N	N	N	N	N	N	Y	2
Distinct Type	2	2	2	2	2	2	2	2	2	2	Y[2]

Notes:

1. The compatibility of datetime values is limited to assignment and comparison:
   - Datetime values can be assigned to string columns and to string variables, as explained in Chapter 2 of *DB2 SQL Reference*.
   - A valid string representation of a date can be assigned to a date column or compared to a date.
   - A valid string representation of a time can be assigned to a time column or compared to a time.
   - A valid string representation of a timestamp can be assigned to a timestamp column or compared to a timestamp.
2. A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see Chapter 2 of *DB2 SQL Reference*.
3. All character strings, even those with subtype FOR BIT DATA, are not compatible with binary strings.
4. These data types are compatible if the graphic string is Unicode UTF-16. On assignment and comparison from Graphic to Character, the resulting length is 3 * (LENGTH(graphic string)).
5. Character strings with subtype FOR BIT DATA are not compatible with Unicode UTF-16 Graphic Data.

Selecting columns: SELECT

You have several options for selecting columns from a database for your result tables. This section describes how to select columns using a variety of techniques.

Selecting all columns: SELECT *

You do not need to know the column names to select DB2 data. Use an asterisk (*) in the SELECT clause to indicate that you want to retrieve from each selected row of the named table.

Example: SELECT *: This SQL statement selects all columns from the department table:

```
SELECT *
  FROM DSN8710.DEPT;
```

The result table looks like this:

```
DEPTNO  DEPTNAME                             MGRNO   ADMRDEPT  LOCATION
======  ==================================   ======  ========  ========
A00     SPIFFY COMPUTER SERVICE DIV.         000010  A00       --------
B01     PLANNING                             000020  A00       --------
C01     INFORMATION CENTER                   000030  A00       --------
D01     DEVELOPMENT CENTER                   ------  A00       --------
D11     MANUFACTURING SYSTEMS                000060  D01       --------
D21     ADMINISTRATION SYSTEMS               000070  D01       --------
E01     SUPPORT SERVICES                     000050  A00       --------
E11     OPERATIONS                           000090  E01       --------
E21     SOFTWARE SUPPORT                     000100  E01       --------
F22     BRANCH OFFICE F2                     ------  E01       --------
G22     BRANCH OFFICE G2                     ------  E01       --------
H22     BRANCH OFFICE H2                     ------  E01       --------
I22     BRANCH OFFICE I2                     ------  E01       --------
J22     BRANCH OFFICE J2                     ------  E01       --------
```

Because the example does not specify a WHERE clause, the statement retrieves data from all rows.

The dashes for MGRNO and LOCATION in the result table indicate null values.

SELECT * is recommended mostly for use with dynamic SQL and view definitions. You can use SELECT * in static SQL, but this is not recommended; if you add a column to the table to which SELECT * refers, the program might reference columns for which you have not defined receiving host variables. For more information on host variables, see "Accessing data using host variables and host structures" on page 67.

If you list the column names in a static SELECT statement instead of using an asterisk, you can avoid the problem just mentioned. You can also see the relationship between the receiving host variables and the columns in the result table.

Selecting some columns: SELECT column-name

Select the column or columns you want by naming each column. All columns appear in the order you specify, not in their order in the table.

Example: SELECT column-name This SQL statement selects only the MGRNO and DEPTNO columns from the department table:

```
SELECT MGRNO, DEPTNO
  FROM DSN8710.DEPT;
```

The result table looks like this:

```
MGRNO   DEPTNO
======  ======
000010  A00
000020  B01
000030  C01
------  D01
000050  E01
000060  D11
000070  D21
```

```
000090  E11
000100  E21
------  F22
------  G22
------  H22
------  I22
------  J22
```

With a single SELECT statement, you can select data from one column or as many as 750 columns.

Selecting DB2 data that is not in a table: Using SYSDUMMY1

DB2 provides an EBCDIC table, SYSIBM.SYSDUMMY1, that you can use to select DB2 data that is not in a table.

For example, if you want to execute a DB2 built-in function on host variable, you can use an SQL statement like this:

```
SELECT RAND(:HRAND)
  FROM SYSIBM.SYSDUMMY1;
```

Selecting derived columns: SELECT expression

You can select columns derived from a constant, an expression, or a function.

Example: SELECT with an expression: This SQL statement generates a result table in which the second column is a derived column that is generated by adding the values of the SALARY, BONUS, and COMM columns.

```
SELECT EMPNO, (SALARY + BONUS + COMM)
  FROM DSN8710.EMP;
```

Derived columns in a result table, such as (SALARY + BONUS + COMM), do not have names. The AS clause lets you give names to unnamed columns. See "Naming result columns: AS" for information on the AS clause.

If you want to order the rows of data in the result table, use the ORDER BY clause described in "Putting the rows in order: ORDER BY" on page 9.

Eliminating duplicate rows: DISTINCT

The DISTINCT keyword removes duplicate rows from your result, so that each row contains unique data.

Example: SELECT DISTINCT: The following SELECT statement lists unique department numbers for administrating departments:

```
SELECT DISTINCT ADMRDEPT
  FROM DSN8710.DEPT;
```

The result table looks like this:

```
ADMRDEPT
========
A00
D01
E01
```

Naming result columns: AS

With AS, you can name result columns in a SELECT clause. This is particularly useful for a column that is derived from an expression or a function. For syntax and more information, see Chapter 2 of *DB2 SQL Reference*.

The following examples show different ways to use the AS clause.

Example: SELECT with AS CLAUSE: The expression SALARY+BONUS+COMM has the name TOTAL_SAL.

```
SELECT SALARY+BONUS+COMM AS TOTAL_SAL
  FROM DSN8710.EMP
  ORDER BY TOTAL_SAL;
```

Example: CREATE VIEW with AS clause: You can specify result column names in the select-clause of a CREATE VIEW statement. You do not need to supply the column list of CREATE VIEW, because the AS keyword names the derived column. The columns in the view EMP_SAL are EMPNO and TOTAL_SAL.

```
CREATE VIEW EMP_SAL AS
  SELECT EMPNO,SALARY+BONUS+COMM AS TOTAL_SAL
      FROM DSN8710.EMP;
```

Example: UNION ALL with AS clause: You can use the AS clause to give the same name to corresponding columns of tables in a union. The third result column from the union of the two tables has the name TOTAL_VALUE, even though it contains data derived from columns with different names:

```
SELECT 'On hand' AS STATUS, PARTNO, QOH * COST AS TOTAL_VALUE
    FROM PART_ON_HAND
UNION ALL
SELECT 'Ordered' AS STATUS, PARTNO, QORDER * COST AS TOTAL_VALUE
    FROM ORDER_PART
ORDER BY PARTNO, TOTAL_VALUE;
```

The column STATUS and the derived column TOTAL_VALUE have the same name in the first and second result tables, and are combined in the union of the two result tables:

```
STATUS       PARTNO   TOTAL_VALUE
-----------  ------   -----------
On hand      00557    345.60
Ordered      00557    150.50
  .
  .
  .
```

For information on unions, see "Merging lists of values: UNION" on page 12.

Example: FROM clause with AS clause: Use the AS clause in a FROM clause to assign a name to a derived column that you want to refer to in a GROUP BY clause. Using the AS clause in the first SELECT clause causes an error, because the names assigned in the AS clause do not yet exist when the GROUP BY executes. However, you can use an AS clause of a subselect in the outer GROUP BY clause, because the subselect is at a lower level than the GROUP BY that references the name. This SQL statement names HIREYEAR in the nested table expression, which lets you use the name of that result column in the GROUP BY clause:

```
SELECT HIREYEAR, AVG(SALARY)
    FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
              FROM DSN8710.EMP) AS NEWEMP
    GROUP BY HIREYEAR;
```

Selecting rows using search conditions: WHERE

Use a WHERE clause to select the rows that meet certain conditions. A WHERE clause specifies a *search condition*. A search condition consists of one or more *predicates*. A predicate specifies a test you want DB2 to apply to each table row.

DB2 evaluates a predicate for a row as true, false, or unknown. Results are unknown only if an operand is null.

If a search condition contains a column of a distinct type, the value to which that column is compared must be of the same distinct type, or you must cast the value to the distinct type. See "Chapter 15. Creating and using distinct types" on page 299 for more information.

The next sections illustrate different comparison operators that you can use in a predicate in a WHERE clause. The following table lists the comparison operators.

Table 2. Comparison operators used in conditions

Type of comparison	Specified with...	Example
Equal to null	IS NULL	PHONENO IS NULL
Equal to	=	DEPTNO = 'X01'
Not equal to	<>	DEPTNO <> 'X01'
Less than	<	AVG(SALARY) < 30000
Less than or equal to	<=	AGE <= 25
Not less than	>=	AGE >= 21
Greater than	>	SALARY > 2000
Greater than or equal to	>=	SALARY >= 5000
Not greater than	<=	SALARY <= 5000
Similar to another value	LIKE	NAME LIKE '%SMITH%' or STATUS LIKE 'N_'
At least one of two conditions	OR	HIREDATE < '1965-01-01' OR SALARY < 16000
Both of two conditions	AND	HIREDATE < '1965-01-01' AND SALARY < 16000
Between two values	BETWEEN	SALARY BETWEEN 20000 AND 40000
Equals a value in a set	IN (X, Y, Z)	DEPTNO IN ('B01', 'C01', 'D01')

You can also search for rows that *do not* satisfy one of the above conditions, by using the NOT keyword before the specified condition.

Putting the rows in order: ORDER BY

To retrieve rows in a specific order, use the ORDER BY clause. Using ORDER BY is the only way to can guarantee that your rows are ordered as you want them. The following sections show you how to use the ORDER BY clause.

Specifying the sort key

The order of the selected rows depends on the sort keys that you identify in the ORDER BY clause. A sort key can be a column name, an integer that represents the number of a column in the result table, or an expression. DB2 orders the rows by the first sort key, followed by the second sort key, and so on.

You can list the rows in ascending or descending order. Null values appear last in an ascending sort and first in a descending sort.

DB2 sorts strings in the collating sequence associated with the encoding scheme of the table. DB2 sorts numbers algebraically and sorts datetime values chronologically.

Example: ORDER BY clause with a column name as the sort key:Retrieve the employee numbers, last names, and hire dates of employees in department A00 in ascending order of hire dates:

```
SELECT EMPNO, LASTNAME, HIREDATE
  FROM DSN8710.EMP
  WHERE WORKDEPT = 'A00'
  ORDER BY HIREDATE ASC;
```

This is the result:

```
EMPNO   LASTNAME         HIREDATE
======  ===============  ==========
000110  LUCCHESI         1958-05-16
000120  O'CONNELL        1963-12-05
000010  HAAS             1965-01-01
200010  HEMMINGER        1965-01-01
200120  ORLANDO          1972-05-05
```

Example: ORDER BY clause with an expression as the sort key:Retrieve the employee numbers, salaries, commissions, and total compensation (salary plus commission) for employees with a total compensation of greater than 40000. Order the results by total compensation:

```
SELECT EMPNO, SALARY, COMM, SALARY+COMM AS "TOTAL COMP"
  FROM DSN8710.EMP
  WHERE SALARY+COMM = 40000
  ORDER BY SALARY+COMM;
```

This is the result:

```
EMPNO     SALARY     COMM   TOTAL COMP
======  ========  =======  ==========
000030  38250.00  3060.00    41310.00
000050  40175.00  3214.00    43389.00
000020  41250.00  3300.00    44550.00
000110  46500.00  3720.00    50220.00
200010  46500.00  4220.00    50720.00
000010  52750.00  4220.00    56970.00
```

Referencing derived columns

If you use the AS clause to name an unnamed column in a SELECT statement, you can use that name in the ORDER BY clause. For example, the following SQL statement orders the selected information by total salary:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
  FROM DSN8710.EMP
  ORDER BY TOTAL_SAL;
```

Summarizing group values: GROUP BY

Use GROUP BY to group rows by the values of one or more columns. You can then apply column functions to each group.

Except for the columns named in the GROUP BY clause, the SELECT statement must specify any other selected columns as an operand of one of the column functions.

The following SQL statement lists, for each department, the lowest and highest education level within that department.

```
SELECT WORKDEPT, MIN(EDLEVEL), MAX(EDLEVEL)
  FROM DSN8710.EMP
  GROUP BY WORKDEPT;
```

If a column you specify in the GROUP BY clause contains null values, DB2 considers those null values to be equal. Thus, all nulls form a single group.

When it is used, the GROUP BY clause follows the FROM clause and any WHERE clause, and precedes the ORDER BY clause.

You can also group the rows by the values of more than one column. For example, the following statement finds the average salary for men and women in departments A00 and C01:

```
SELECT WORKDEPT, SEX, AVG(SALARY) AS AVG_SALARY
  FROM DSN8710.EMP
  WHERE WORKDEPT IN ('A00', 'C01')
  GROUP BY WORKDEPT, SEX;
```

gives this result:

```
WORKDEPT  SEX  AVG_SALARY
========  ===  ===========
A00       F     49625.00000000
A00       M     35000.00000000
C01       F     29722.50000000
```

DB2 groups the rows first by department number and next (within each department) by sex before DB2 derives the average SALARY value for each group.

Subjecting groups to conditions: HAVING

Use HAVING to specify a search condition that each retrieved group must satisfy. The HAVING clause acts like a WHERE clause for groups, and can contain the same kind of search conditions you can specify in a WHERE clause. The search condition in the HAVING clause tests properties of each group rather than properties of individual rows in the group.

This SQL statement:

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY
  FROM DSN8710.EMP
  GROUP BY WORKDEPT
  HAVING COUNT(*) > 1
  ORDER BY WORKDEPT;
```

gives this result:

```
WORKDEPT   AVG_SALARY
========   ===============
A00        40850.00000000
C01        29722.50000000
D11        25147.27272727
D21        25668.57142857
E11        21020.00000000
E21        24086.66666666
```

Compare the preceding example with the second example shown in "Summarizing group values: GROUP BY" on page 10. The HAVING COUNT(*) > 1 clause ensures

that only departments with more than one member display. (In this case, departments B01 and E01 do not display.)

The HAVING clause tests a property of the group. For example, you could use it to retrieve the average salary and minimum education level of women in each department in which all female employees have an education level greater than or equal to 16. Assuming you only want results from departments A00 and D11, the following SQL statement tests the group property, MIN(EDLEVEL):

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY,
  MIN(EDLEVEL) AS MIN_EDLEVEL
  FROM DSN8710.EMP
  WHERE SEX = 'F' AND WORKDEPT IN ('A00', 'D11')
  GROUP BY WORKDEPT
  HAVING MIN(EDLEVEL) >= 16;
```

The SQL statement above gives this result:

```
WORKDEPT  AVG_SALARY       MIN_EDLEVEL
========  ===============  ============
A00       49625.00000000   18
D11       25817.50000000   17
```

When you specify both GROUP BY and HAVING, the HAVING clause must follow the GROUP BY clause. A function in a HAVING clause can include DISTINCT if you have not used DISTINCT anywhere else in the same SELECT statement. You can also connect multiple predicates in a HAVING clause with AND and OR, and you can use NOT for any predicate of a search condition.

Merging lists of values: UNION

Using the UNION keyword, you can combine two or more SELECT statements to form a single result table. When DB2 encounters the UNION keyword, it processes each SELECT statement to form an interim result table, and then combines the interim result table of each statement. If you use UNION to combine two columns with the same name, the result table inherits that name.

When you use the UNION statement, the SQLNAME field of the SQLDA contains the column names of the first operand.

Using UNION to eliminate duplicates

You can use UNION to eliminate duplicates when merging lists of values obtained from several tables. For example, you can obtain a combined list of employee numbers that includes both of the following:
• People in department D11
• People whose assignments include projects MA2112, MA2113, and AD3111.

For example, this SQL statement:

```
SELECT EMPNO
  FROM DSN8710.EMP
  WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
  FROM DSN8710.EMPPROJACT
  WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
  ORDER BY EMPNO;
```

gives a combined result table containing employee numbers in ascending order with no duplicates listed.

If you have an ORDER BY clause, it must appear after the last SELECT statement that is part of the union. In this example, the first column of the final result table determines the final order of the rows.

Using UNION ALL to keep duplicates

If you want to keep duplicates in the result of a UNION, specify the optional keyword ALL after the UNION keyword.

This SQL statement:

```
SELECT EMPNO
  FROM DSN8710.EMP
  WHERE WORKDEPT = 'D11'
UNION ALL
SELECT EMPNO
  FROM DSN8710.EMPPROJACT
  WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
  ORDER BY EMPNO;
```

gives a combined result table containing employee numbers in ascending order, and includes duplicate numbers.

Using 15-digit and 31-digit precision for decimal numbers

DB2 allows two sets of rules for determining the precision and scale of the result of an operation with decimal numbers.

- DEC15 rules allow a maximum precision of 15 digits in the result of an operation. Those rules are in effect when both operands have precisions of 15 or less, unless one of the circumstances that imply DEC31 rules applies.

- DEC31 rules allow a maximum precision of 31 digits in the result. Those rules are in effect if any of the following is true:

  - Either operand of the operation has a precision greater than 15.

  - The operation is in a dynamic SQL statement, and any of the following conditions is true:

    - The current value of special register CURRENT PRECISION is DEC31.

    - The installation option for DECIMAL ARITHMETIC on panel DSNTIPF is DEC31 or 31, the installation option for USE FOR DYNAMICRULES on panel DSNTIPF is YES, and the value of CURRENT PRECISION has not been set by the application.

    - The SQL statement has bind, define, or invoke behavior, the statement is in an application precompiled with option DEC(31), the installation option for USE FOR DYNAMICRULES on panel DSNTIPF is NO, and the value of CURRENT PRECISION has not been set by the application. See "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415 for an explanation of bind, define, and invoke behavior.

  - The operation is in an embedded (static) SQL statement that you precompiled with the DEC(31) option, or with the default for that option when the install option DECIMAL ARITHMETIC is DEC31 or 31. (See "Step 1: Process SQL statements" on page 396 for a information on precompiling and a list of all precompiler options.)

The choice of whether to use DEC15 or DEC31 is a trade-off:

- Choose DEC15 to avoid an error when the calculated scale of the result of a simple multiply or divide operation is less than 0. Although this error can occur with either set of rules, it is more common with DEC31 rules.
- Choose DEC31 to reduce the chance of overflow, or when dealing with precisions greater than 15.

Avoiding decimal arithmetic errors: For static SQL statements, the simplest way to avoid a division error is to override DEC31 rules by specifying the precompiler option DEC(15). That reduces the probability of errors for statements embedded in the program.

If the dynamic SQL statements have bind, define, or invoke behavior and the value of the installation option for USE FOR DYNAMICRULES on panel DSNTIPF is NO, you can use the precompiler option DEC(15) to override DEC31 rules.

For a dynamic statement, or for a single static statement, use the scalar function DECIMAL to specify values of the precision and scale for a result that causes no errors.

For a dynamic statement, before you execute the statement, set the value of special register CURRENT PRECISION to DEC15.

Even if you use DEC31 rules, multiplication operations can sometimes cause overflow because the precision of the product is greater than 31. To avoid overflow from multiplication of large numbers, use the MULTIPLY_ALT built-in function instead of the multiplication operator.

Finding information in the DB2 catalog

The examples below show you how to access the DB2 system catalog tables to:
- List the tables that you can access
- List the column names of a table

The contents of the DB2 system catalog tables can be a useful reference tool when you begin to develop an SQL statement or an application program.

Displaying a list of tables you can use

The catalog table, SYSIBM.SYSTABAUTH, lists table privileges granted to authorization IDs. To display the tables that you have authority to access (by privileges granted either to your authorization ID or to PUBLIC), you can execute an SQL statement like that shown in the following example. To do this, you must have the SELECT privilege on SYSIBM.SYSTABAUTH.

```
SELECT DISTINCT TCREATOR, TTNAME
  FROM SYSIBM.SYSTABAUTH
  WHERE GRANTEE IN (USER, 'PUBLIC', 'PUBLIC*') AND GRANTEETYPE = ' ';
```

If your DB2 subsystem uses an exit routine for access control authorization, you cannot rely on catalog queries to tell you what tables you can access. When such an exit routine is installed, RACF as well as DB2 control table access.

Displaying a list of columns in a table

Another catalog table, SYSIBM.SYSCOLUMNS, describes every column of every table. Suppose you execute the previous example (displaying a list of tables you can access) and now want to display information about table DSN8710.DEPT. To execute the following example, you must have the SELECT privilege on SYSIBM.SYSCOLUMNS.

```
SELECT NAME, COLTYPE, SCALE, LENGTH
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBNAME = 'DEPT'
  AND TBCREATOR = 'DSN8710';
```

If the table about which you display column information includes LOB or ROWID columns, the LENGTH field for those columns contains the number of bytes those column occupy in the base table, rather than the length of the LOB or ROWID data. To determine the maximum length of data for a LOB or ROWID column, include the LENGTH2 column in your query. For example:

```
SELECT NAME, COLTYPE, LENGTH, LENGTH2
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBNAME = 'EMP_PHOTO_RESUME'
  AND TBCREATOR = 'DSN8710';
```

Chapter 2. Working with tables and modifying data

This chapter discusses these topics:
- Creating your own tables: CREATE TABLE
- "Working with temporary tables" on page 19
- "Dropping tables: DROP TABLE" on page 23
- "Defining a view: CREATE VIEW" on page 23
- "Changing data through a view" on page 24
- "Dropping views: DROP VIEW" on page 25
- "Inserting a row: INSERT" on page 25
- "Updating current values: UPDATE" on page 29
- "Deleting rows: DELETE" on page 30

See *DB2 SQL Reference* for more information about working with tables and data.

Working with tables

You might need to create or drop the tables that you are working with. You might create new tables, copy existing tables, add columns, add or drop referential and check constraints, or make any number of changes. This section discusses how to create and work with tables.

Creating your own tables: CREATE TABLE

Use the CREATE TABLE statement to create a table. The following SQL statement creates a table named PRODUCT:

```
CREATE TABLE PRODUCT
  (SERIAL       CHAR(8)     NOT NULL,
   DESCRIPTION  VARCHAR(60) DEFAULT,
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        DEFAULT);
```

The elements of the CREATE statement are:
- CREATE TABLE, which names the table PRODUCT.
- A list of the columns that make up the table. For each column, specify:
  - The column's name (for example, SERIAL).
  - The data type and length attribute (for example, CHAR(8)). For further information about data types, see "Data types" on page 3.
- The encoding scheme for the table.

  Specify CCSID EBCDIC to use an EBCDIC encoding scheme, CCSID ASCII to use an ASCII encoding scheme, or CCSID UNICODE to use a Unicode encoding scheme. If the CREATE TABLE statement does not have a LIKE clause, the default is the encoding scheme of the table space in which the table resides. If the CREATE TABLE statement has a LIKE clause, the default CCSID is the CCSID of the table in the LIKE clause. See "Creating work tables" on page 18 for examples of using the LIKE clause.
- Optionally, a default value. See "Identifying defaults" on page 18.
- Optionally, a referential constraint or table check constraint. See "Using referential constraints" on page 203 and "Using table check constraints" on page 201.

17

Identifying defaults

If you want to constrain the inputs or identify the defaults, you can describe the columns using:

- NOT NULL, when the column cannot contain null values.
- UNIQUE, when the value for each row must be unique, and the column cannot contain null values.
- DEFAULT, when the column has one of the following DB2-assigned defaults:
  - For numeric fields, zero is the default value.
  - For fixed-length strings, blank is the default value.
  - For variable-length strings, including LOB strings, the empty string (string of zero-length) is the default value.
  - For datetime fields, the current value of the associated special register is the default value.
- DEFAULT *value*, when you want to identify one of the following as the default value:
  - A constant
  - USER, which uses the run-time value of the USER special register
  - CURRENT SQLID, which uses the SQL authorization ID of the process
  - NULL
  - The name of a cast function, to cast a default value to the distinct type of a column

You must separate each column description from the next with a comma, and enclose the entire list of column descriptions in parentheses.

Creating work tables

Before testing SQL statements that insert, update, and delete rows, you should create *work tables* (duplicates of the DSN8710.EMP and DSN8710.DEPT tables), so that the original sample tables remain intact. This section shows how to create two work tables and how to fill a work table with the contents of another table.

Each example shown in this chapter assumes you logged on using your own authorization ID. The authorization ID qualifies the name of each object you create. For example, if your authorization ID is SMITH, and you create table YDEPT, the name of the table is SMITH.YDEPT. If you want to access table DSN8710.DEPT, you must refer to it by its complete name. If you want to access your own table YDEPT, you need only to refer to it as "YDEPT".

Creating a new department table

Use the following statements to create a new department table called YDEPT, modeled after an existing table called DSN8710.DEPT, and an index for YDEPT:

```
CREATE TABLE YDEPT
  LIKE DSN8710.DEPT;

CREATE UNIQUE INDEX YDEPTX
  ON YDEPT (DEPTNO);
```

If you want DEPTNO to be a primary key as in the sample table, explicitly define the key. Use an ALTER TABLE statement:

```
ALTER TABLE YDEPT
  PRIMARY KEY(DEPTNO);
```

You can use an INSERT statement with a SELECT clause to copy rows from one table to another. The following statement copies all of the rows from DSN8710.DEPT to your own YDEPT work table.

```
INSERT INTO YDEPT
  SELECT *
    FROM DSN8710.DEPT;
```

For information on the INSERT statement, see "Modifying DB2 data" on page 25.

Creating a new employee table
You can use the following statements to create a new employee table called YEMP.

```
CREATE TABLE YEMP
      (EMPNO     CHAR(6)        PRIMARY KEY NOT NULL,
       FIRSTNME  VARCHAR(12)    NOT NULL,
       MIDINIT   CHAR(1)        NOT NULL,
       LASTNAME  VARCHAR(15)    NOT NULL,
       WORKDEPT  CHAR(3)        REFERENCES YDEPT
                                  ON DELETE SET NULL,
       PHONENO   CHAR(4)        UNIQUE NOT NULL,
       HIREDATE  DATE                 ,
       JOB       CHAR(8)              ,
       EDLEVEL   SMALLINT             ,
       SEX       CHAR(1)              ,
       BIRTHDATE DATE                 ,
       SALARY    DECIMAL(9, 2)        ,
       BONUS     DECIMAL(9, 2)        ,
       COMM      DECIMAL(9, 2)      );
```

This statement also creates a referential constraint between the foreign key in YEMP (WORKDEPT) and the primary key in YDEPT (DEPTNO). It also restricts all phone numbers to unique numbers.

If you want to change a table definition after you create it, use the statement ALTER TABLE.

If you want to change a table name after you create it, use the statement RENAME TABLE. For details on the ALTER TABLE and RENAME TABLE statements, see Chapter 5 of *DB2 SQL Reference*. You cannot drop a column from a table or change a column definition. However, you can add and drop constraints on columns in a table.

Working with temporary tables

When you need a table only for the life of an application process, you can create a temporary table. There are two kinds of temporary tables:
- Created temporary tables, which you define using a CREATE GLOBAL TEMPORARY TABLE statement
- Declared temporary tables, which you define using a DECLARE GLOBAL TEMPORARY TABLE statement

SQL statements that use temporary tables can run faster because:
- DB2 does no logging (for created temporary tables) or limited logging (for declared temporary tables).
- DB2 does not locking (for created temporary tables) or limited locking (for declared temporary tables).

Temporary tables are especially useful when you need to sort or query intermediate result tables that contain large numbers of rows, but you want to store only a small subset of those rows permanently.

Temporary tables can also return result sets from stored procedures. For more information, see "Writing a stored procedure to return result sets to a DRDA client" on page 545. The following sections provide more details on created temporary tables and declared temporary tables.

Working with created temporary tables

You create the *definition* of a created temporary table using the SQL statement CREATE GLOBAL TEMPORARY TABLE.

Example: This statement creates the definition of a table called TEMPPROD:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD
  (SERIAL       CHAR(8)     NOT NULL,
   DESCRIPTION  VARCHAR(60) NOT NULL,
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        NOT NULL);
```

Example:

You can also create a definition by copying the definition of a base table:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD LIKE PROD;
```

The SQL statements in the previous examples create identical definitions, even though table PROD contains two columns, DESCRIPTION and CURDATE, that are defined as NOT NULL WITH DEFAULT. Because created temporary tables do not support WITH DEFAULT, DB2 changes the definitions of DESCRIPTION and CURDATE to NOT NULL when you use the second method to define TEMPPROD.

After you execute one of the two CREATE statements, the definition of TEMPPROD exists, but no instances of the table exist. To drop the definition of TEMPPROD, you must execute this statement:

```
DROP TABLE TEMPPROD;
```

To create an instance of TEMPPROD, you must use TEMPPROD in an application. DB2 creates an instance of the table when TEMPPROD appears in one of these SQL statements:
- OPEN
- SELECT
- INSERT
- DELETE

An instance of a created temporary table exists at the current server until one of the following actions occurs:
- The remote server connection under which the instance was created terminates.
- The unit of work under which the instance was created completes.

  When you execute a ROLLBACK statement, DB2 deletes the instance of the created temporary table. When you execute a COMMIT statement, DB2 deletes the instance of the created temporary table unless a cursor for accessing the created temporary table is defined WITH HOLD and is open.
- The application process ends.

For example, suppose that you create a definition of TEMPPROD and then run an application that contains these statements:

```
EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM TEMPPROD;
EXEC SQL INSERT INTO TEMPPROD SELECT * FROM PROD;
EXEC SQL OPEN C1;
  :
  :
EXEC SQL COMMIT;
  :
  :
EXEC SQL CLOSE C1;
```

When you execute the INSERT statement, DB2 creates an instance of TEMPPROD
and populates that instance with rows from table PROD. When the COMMIT
statement is executed, DB2 deletes all rows from TEMPPROD. If, however, you
change the declaration of C1 to:

```
EXEC SQL DECLARE C1 CURSOR WITH HOLD
  FOR SELECT * FROM TEMPPROD;
```

DB2 does not delete the contents of TEMPPROD until the application ends
because C1, a cursor defined WITH HOLD, is open when the COMMIT statement is
executed. In either case, DB2 drops the instance of TEMPPROD when the
application ends.

Working with declared temporary tables

You create an instance of a declared temporary table using the SQL statement
DECLARE GLOBAL TEMPORARY TABLE. That instance is known only to the
application process in which the table is declared, so you can declare temporary
tables with the same name in different applications.

Before you can define declared temporary tables, you must create a special
database and table spaces for them. You do that by executing the CREATE
DATABASE statement with the AS TEMP clause, and then creating segmented
table spaces in that database. A DB2 subsystem can have only one database for
declared temporary tables, but that database can contain more than one table
space.

Example: These statements create a database and table space for declared
temporary tables:

```
CREATE DATABASE DTTDB AS TEMP;
CREATE TABLESPACE DTTTS IN DTTDB
  SEGSIZE 4;
```

You can define a declared temporary table in any of the following ways:

- Specify all the columns in the table.

  Unlike columns of created temporary tables, columns of declared temporary
  tables can include the WITH DEFAULT clause.

- Use a LIKE clause to copy the definition of a base table, created temporary
  table, or view.

  If the base table or created temporary table that you copy has identity columns,
  you can specify that the corresponding columns in the declared temporary table
  are also identity columns. Do that by specifying the INCLUDING IDENTITY
  COLUMN ATTRIBUTES clause when you define the declared temporary table.

- Use a fullselect to choose specific columns from a base table, created temporary
  table, or view.

  If the base table, created temporary table, or view from which you select columns
  has identity columns, you can specify that the corresponding columns in the
  declared temporary table are also identity columns. Do that by specifying the
  INCLUDING IDENTITY COLUMN ATTRIBUTES clause when you define the
  declared temporary table.

If you want the declared temporary table columns to inherit the defaults for columns of the table or view that is named in the fullselect, specify the INCLUDING COLUMN DEFAULTS clause. If you want the declared temporary table columns to have default values that correspond to their data types, specify the USING TYPE DEFAULTS clause.

Example: This statement defines a declared temporary table called TEMPPROD by explicitly specifying the columns.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  (SERIAL       CHAR(8)     NOT NULL WITH DEFAULT '99999999',
   DESCRIPTION  VARCHAR(60) NOT NULL,
   PRODCOUNT    INTEGER GENERATED ALWAYS AS IDENTITY
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        NOT NULL);
```

Example: This statement defines a declared temporary table called TEMPPROD by copying the definition of a base table. The base table has an identity column that the declared temporary table also uses as an identity column.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD LIKE BASEPROD
  INCLUDING IDENTITY COLUMN ATTRIBUTES;
```

Example: This statement defines a declared temporary table called TEMPPROD by selecting columns from a view. The view has an identity column that the declared temporary table also uses as an identity column. The declared temporary table inherits the default column values from the view definition.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  AS (SELECT * FROM PRODVIEW)
  DEFINITION ONLY
  INCLUDING IDENTITY COLUMN ATTRIBUTES
  INCLUDING COLUMN DEFAULTS;
```

After you execute a DECLARE GLOBAL TEMPORARY TABLE statement, the definition of the declared temporary table exists as long as the application process runs. If you need to delete the definition before the application process completes, you can do that with the DROP TABLE statement. For example, to drop the definition of TEMPPROD, execute this statement:

```
DROP TABLE SESSION.TEMPPROD;
```

DB2 creates an empty instance of a declared temporary table when it executes the DECLARE GLOBAL TEMPORARY TABLE statement. You can populate the declared temporary table using INSERT statements, modify the table using searched or positioned UPDATE or DELETE statements, and query the table using SELECT statements. You can also create indexes on the declared temporary table.

When you execute a COMMIT statement in an application with a declared temporary table, DB2 deletes all the rows from the table or keeps the rows, depending on the ON COMMIT clause that you specify in the DECLARE GLOBAL TEMPORARY TABLE statement. ON COMMIT DELETE ROWS, which is the default, causes all rows to be deleted from the table at a commit point, unless there is a held cursor open on the table at the commit point.. ON COMMIT PRESERVE ROWS causes the rows to be remain past the commit point.

For example, suppose that you execute these statement in an application program:

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  AS (SELECT * FROM BASEPROD)
  DEFINITION ONLY
  INCLUDING IDENTITY COLUMN ATTRIBUTES
  INCLUDING COLUMN DEFAULTS
  ON COMMIT PRESERVE ROWS;
EXEC SQL INSERT INTO SESSION.TEMPPROD SELECT * FROM BASEPROD;
 .
 .
 .
EXEC SQL COMMIT;
 .
 .
 .
```

When DB2 executes the DECLARE GLOBAL TEMPORARY TABLE statement, DB2 creates an empty instance of TEMPPROD. The INSERT statement populates that instance with rows from table BASEPROD. The qualifier, SESSION, must be specified in any statement that references TEMPPROD. When DB2 executes the COMMIT statement, DB2 keeps all rows in TEMPPROD because TEMPPROD is defined with ON COMMIT PRESERVE ROWS. When the program ends, DB2 drops TEMPPROD.

Dropping tables: DROP TABLE

This SQL statement drops the YEMP table:

```
DROP TABLE YEMP;
```

Use the DROP TABLE statement with care: Dropping a table is NOT equivalent to deleting all its rows. When you drop a table, you lose more than both its data and its definition. You lose all synonyms, views, indexes, and referential and check constraints associated with that table. You also lose all authorities granted on the table.

For more information on the DROP statement, see Chapter 5 of *DB2 SQL Reference*.

Working with views

This section discusses how to use CREATE VIEW and DROP VIEW to control your view of existing tables. Although you cannot modify an existing view, you can drop it and create a new one if your base tables change in a way that affects the view. Dropping and creating views does not affect the base tables or their data.

Defining a view: CREATE VIEW

A *view* does not contain data; it is a stored definition of a set of rows and columns. A view can present any or all of the data in one or more tables, and, in most cases, is interchangeable with a table. Using views can simplify writing SQL statements.

Use the CREATE VIEW statement to define a view and give the view a name, just as you do for a table.

```
CREATE VIEW VDEPTM AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
    FROM DSN8710.DEPT, DSN8710.EMP
      WHERE DSN8710.EMP.EMPNO = DSN8710.DEPT.MGRNO;
```

This view shows each department manager's name with the department data in the DSN8710.DEPT table.

When a program accesses the data defined by a view, DB2 uses the view definition to return a set of rows the program can access with SQL statements. Now that the

view VDEPTM exists, you can retrieve data using the view. To see the departments administered by department D01 and the managers of those departments, execute the following statement:

```
SELECT DEPTNO, LASTNAME
  FROM VDEPTM
    WHERE ADMRDEPT = 'D01';
```

When you create a view, you can reference the USER and CURRENT SQLID special registers in the CREATE VIEW statement. When referencing the view, DB2 uses the value of the USER or CURRENT SQLID that belongs to the user of the SQL statement (SELECT, UPDATE, INSERT, or DELETE) rather than the creator of the view. In other words, a reference to a special register in a view definition refers to its run-time value.

A column in a view might be based on a column in a base table that is an identity column. The column in the view is also an identity column, *except* under any of the following circumstances:
- The column appears more than once in the view.
- The view is based on a join of two or more tables.
- Any column in the view is derived from an expression that refers to an identity column.

You can use views to limit access to certain kinds of data, such as salary information. You can also use views to do the following:
- Make a subset of a table's data available to an application. For example, a view based on the employee table might contain rows for a particular department only.
- Combine columns from two or more tables and make the combined data available to an application. By using a SELECT statement that matches values in one table with those in another table, you can create a view that presents data from both tables. However, you can only *select* data from this type of view. *You cannot update, delete, or insert data using a view that joins two or more tables.*
- Combine rows from two or more tables and make the combined data available to an application. By using two or more subselects that are connected by UNION or UNION ALL operators, you can create a view that presents data from several tables. However, you can only *select* data from this type of view. *You cannot update, delete, or insert data using a view that contains UNION operations.*
- Present computed data, and make the resulting data available to an application. You can compute such data using any function or operation that you can use in a SELECT statement.

Changing data through a view

Some views are read-only, while others are subject to update or insert restrictions. (See Chapter 5 of *DB2 SQL Reference* for more information about read-only views.) If a view does not have update restrictions, there are some additional things to consider:
- You must have the appropriate authorization to insert, update, or delete rows using the view.
- When you use a view to insert a row into a table, the view definition must specify all the columns in the base table that do not have a default value. The row being inserted must contain a value for each of those columns.
- Views that you can use to update data are subject to the same referential constraints and table check constraints as the tables you used to define the views.

Dropping views: DROP VIEW

When you drop a view, you also drop all views defined on that view. This SQL statement drops the VDEPTM view:

```
DROP VIEW VDEPTM;
```

Modifying DB2 data

This section discusses how to add or modify data in an existing table using the statements INSERT, UPDATE, and DELETE.

Inserting a row: INSERT

Use an INSERT statement to add new rows to a table or view. Using an INSERT statement, you can do the following:

- Specify the values to insert in a single row. You can specify constants, host variables, expressions, DEFAULT, or NULL.
- Include a SELECT statement in the INSERT statement to tell DB2 that another table or view contains the data for the new row (or rows). "Filling a table from another table: Mass INSERT" on page 26, explains how to use the SELECT statement within an INSERT statement to add rows to a table.

In either case, for every row you insert, you must provide a value for any column that does not have a default value. For a column that meets one of these conditions, you can specify DEFAULT to tell DB2 to insert the default value for that column:

- Is nullable.
- Is defined with a default value.
- Has data type ROWID. ROWID columns always have default values.
- Is an identity column. Identity columns always have default values.

The values that you can insert into a ROWID column or identity column depend on whether the column is defined with GENERATED ALWAYS or GENERATED BY DEFAULT. See "Inserting data into a ROWID column" on page 27 and "Inserting data into an identity column" on page 27 for more information.

You can name all columns for which you are providing values. Alternatively, you can omit the column name list.

For static insert statements, it is a good idea to name all columns for which you are providing values because:

- Your insert statement is independent of the table format. (For example, you do not have to change the statement when a column is added to the table.)
- You can verify that you are giving the values in order.
- Your source statements are more self-descriptive.

If you do not name the columns in a static insert statement, and a column is added to the table being inserted into, an error can occur if the insert statement is rebound. An error will occur after any rebind of the insert statement unless you change the insert statement to include a value for the new column. This is true, even if the new column has a default value.

When you list the column names, you must specify their corresponding values in the same order as in the list of column names.

For example,
```
INSERT INTO YDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
  VALUES ('E31', 'DOCUMENTATION', '000010', 'E01', ' ');
```

After inserting a new department row into your YDEPT table, you can use a SELECT statement to see what you have loaded into the table. This SQL statement:
```
SELECT *
  FROM YDEPT
  WHERE DEPTNO LIKE 'E%'
  ORDER BY DEPTNO;
```

shows you all the new department rows that you have inserted:

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
E01	SUPPORT SERVICES	000050	A00	-----------
E11	OPERATIONS	000090	E01	-----------
E21	SOFTWARE SUPPORT	000100	E01	-----------
E31	DOCUMENTATION	000010	E01	-----------

There are other ways to enter data into tables:

- You can copy one table into another, as explained in "Filling a table from another table: Mass INSERT".
- You can write an application program to enter large amounts of data into a table. For details, see "Part 2. Coding SQL in your host application program" on page 61.
- You can use the DB2 LOAD utility to enter data from other sources. See Part 2 of *DB2 Utility Guide and Reference* for more information about the LOAD utility.

Filling a table from another table: Mass INSERT
Use a fullselect within an INSERT statement to select rows from one table to insert into another table.

This SQL statement creates a table named TELE:
```
CREATE TABLE TELE
  (NAME2  VARCHAR(15)  NOT NULL,
   NAME1  VARCHAR(12)  NOT NULL,
   PHONE  CHAR(4));
```

This statement copies data from DSN8710.EMP into the newly created table:
```
INSERT INTO TELE
  SELECT LASTNAME, FIRSTNME, PHONENO
    FROM DSN8710.EMP
    WHERE WORKDEPT = 'D21';
```

The two previous statements create and fill a table, TELE, that looks like this:

NAME2	NAME1	PHONE
PULASKI	EVA	7831
JEFFERSON	JAMES	2094
MARINO	SALVATORE	3780
SMITH	DANIEL	0961
JOHNSON	SYBIL	8953
PEREZ	MARIA	9001
MONTEVERDE	ROBERT	3780

The CREATE TABLE statement example creates a table which, at first, is empty. The table has columns for last names, first names, and phone numbers, but does not have any rows.

The INSERT statement fills the newly created table with data selected from the DSN8710.EMP table: the names and phone numbers of employees in Department D21.

Example: The following CREATE statement creates a table that contains an employee's department name as well as the phone number. The fullselect fills the DLIST table with data from rows selected from two existing tables, DSN8710.DEPT and DSN8710.EMP.

```
CREATE TABLE DLIST
  (DEPT    CHAR(3)      NOT NULL,
   DNAME   VARCHAR(36)           ,
   LNAME   VARCHAR(15)  NOT NULL,
   FNAME   VARCHAR(12)  NOT NULL,
   INIT    CHAR                  ,
   PHONE   CHAR(4) );

INSERT INTO DLIST
  SELECT DEPTNO, DEPTNAME, LASTNAME, FIRSTNME, MIDINIT, PHONENO
    FROM DSN8710.DEPT, DSN8710.EMP
    WHERE DEPTNO = WORKDEPT;
```

Inserting data into a ROWID column

Before you insert data into a ROWID column, you must know how the ROWID column is defined. ROWID columns can be defined as GENERATED ALWAYS or GENERATED BY DEFAULT. GENERATED ALWAYS means that DB2 generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and DB2 provides a default value if you do not supply one. For example, suppose that tables T1 and T2 have two columns: an integer column and a ROWID column. For the following statement to execute successfully, ROWIDCOL2 must be defined as GENERATED BY DEFAULT.

```
INSERT INTO T2 (INTCOL2,ROWIDCOL2)
  SELECT INTCOL1, ROWIDCOL1 FROM T1;
```

If ROWIDCOL2 is defined as GENERATED ALWAYS, you cannot insert the ROWID column data from T1 into T2, but you can insert the integer column data. To insert only the integer data, use one of the following methods:

- Specify only the integer column in your INSERT statement:

```
INSERT INTO T2 (INTCOL2)
  SELECT INTCOL1 FROM T1;
```

- Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell DB2 to ignore any values that you supply for system-generated columns:

```
INSERT INTO T2 (INTCOL2,ROWIDCOL2) OVERRIDING USER VALUE
  SELECT INTCOL1, ROWIDCOL1 FROM T1;
```

Inserting data into an identity column

An identity column is a numeric column with ascending or descending values. For an identity column to be the most useful, those values should also be unique.

An identity column is defined in a CREATE TABLE or ALTER TABLE statement. The column has a SMALLINT, INTEGER, or DECIMAL(p,0) data type and is defined with the AS IDENTITY clause. The AS IDENTITY clause specifies that the column is an identity column. The column is also defined with the GENERATED ALWAYS or GENERATED BY DEFAULT clause. GENERATED ALWAYS means that DB2

generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and DB2 provides a default value if you do not supply one.

Before you insert data into an identity column, you must know whether the column is defined as GENERATED ALWAYS or GENERATED BY DEFAULT. If you try to insert a value into an identity column that is defined as GENERATED ALWAYS, the insert operation fails.

The values that DB2 generates for an identity column depend on how the column is defined. The START WITH parameter determines the first value that DB2 generates. The MINVALUE and MAXVALUE parameters determine the minimum and maximum values that DB2 generates. The CYCLE or NO CYCLE parameter determines whether DB2 wraps values when it has generated all values between the START WITH value and MAXVALUE, if the values are ascending, or between the START WITH value and MINVALUE, if the values are descending.

Identity columns that are defined with GENERATED ALWAYS and NO CYCLE are guaranteed to have unique values. For identity columns that are defined as GENERATED BY DEFAULT and NO CYCLE, only the values that DB2 generates are guaranteed to be unique among each other. To guarantee unique values in an identity column, you need to create a unique index on the identity column.

Example: Inserting into an identity column that is defined with GENERATED BY DEFAULT: Suppose that tables T1 and T2 have two columns: a character column and an integer column that is defined as an identity column. For the following statement to execute successfully, IDENTCOL2 must be defined as GENERATED BY DEFAULT.

```
INSERT INTO T2 (CHARCOL2,IDENTCOL2)
  SELECT CHARCOL1, IDENTCOL1 FROM T1;
```

If IDENTCOL2 is defined as GENERATED ALWAYS, you cannot insert the identity column data from T1 into T2, but you can insert the character column data. To insert only the character data, use one of the following methods:

- Specify only the character column in your INSERT statement:

  ```
  INSERT INTO T2 (CHARCOL2)
    SELECT CHARCOL1 FROM T1;
  ```

- Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell DB2 to ignore any values that you supply for system-generated columns:

  ```
  INSERT INTO T2 (CHARCOL2,IDENTCOL2) OVERRIDING USER VALUE
    SELECT CHARCOL1, IDENTCOL1 FROM T1;
  ```

Example: Inserting into an identity column that is defined with CYCLE: Suppose that table T1 is defined like this:

```
CREATE TABLE T1
  (CHARCOL1 CHAR(1),
   IDENTCOL1 SMALLINT GENERATED ALWAYS AS IDENTITY
     (START WITH -1,
      INCREMENT BY 1,
      CYCLE,
      MINVALUE -3,
      MAXVALUE 3));
```

Now suppose that you execute the following INSERT statement six times:

```
INSERT INTO T1 (CHARCOL1) VALUES ('A');
```

When DB2 generates values for IDENTCOL1, it starts with -1 and increments by 1 until it reaches the MAXVALUE of 3 on the fifth INSERT. To generate the value for the sixth INSERT, DB2 cycles back to MINVALUE, which is -3. T1 looks like this after the six INSERTs are executed:

```
CHARCOL1  IDENTCOL1
========  =========
A                -1
A                 0
A                 1
A                 2
A                 3
A                -3
```

Using an INSERT statement in an application program

If DB2 finds an error while executing the INSERT statement, it inserts nothing into the table, and sets error codes in the SQLCODE and SQLSTATE host variables or corresponding fields of the SQLCA. If the INSERT statement is successful, SQLERRD(3) is set to the number of rows inserted. See Appendix C of *DB2 SQL Reference* for more information.

Examples: This statement inserts information about a new employee into the YEMP table. Because YEMP has a foreign key WORKDEPT referencing the primary key DEPTNO in YDEPT, the value inserted for WORKDEPT (E31) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
  VALUES ('000400', 'RUTHERFORD', 'B', 'HAYES', 'E31',
    '5678', '1983-01-01', 'MANAGER', 16, 'M', '1943-07-10', 24000,
    500, 1900);
```

The following statement also inserts a row into the YEMP table. However, the statement does not specify a value for every column. Because the unspecified columns allow nulls, DB2 inserts null values into the columns not specified. Because YEMP has a foreign key WORKDEPT referencing the primary key DEPTNO in YDEPT, the value inserted for WORKDEPT (D11) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
  (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, JOB)
  VALUES ('000410', 'MILLARD', 'K', 'FILLMORE', 'D11', '4888', 'MANAGER');
```

Updating current values: UPDATE

To change the data in a table, use the UPDATE statement. You can also use the UPDATE statement to delete a value from a row's column (without removing the row) by changing the column's value to NULL.

For example, suppose an employee relocates. To update several items of the employee's data in the YEMP work table to reflect the move, you can execute:

```
UPDATE YEMP
  SET JOB = 'MANAGER ',
  PHONENO ='5678'
  WHERE EMPNO = '000400';
```

You cannot update rows in a created temporary table, but you can update rows in a declared temporary table.

The SET clause names the columns that you want to update and provides the values you want to assign to those columns. You can replace a column value with any of the following items:

- A null value

  The column to which you assign the null value must not be defined as NOT NULL.
- An expression

  An expression can be any of the following items:
  - A column
  - A constant
  - A fullselect that returns a scalar or a row
  - A host variable
  - A special register

Next, identify the rows to update:

- To update a single row, use a WHERE clause that locates one, and only one, row
- To update several rows, use a WHERE clause that locates only the rows you want to update.

If you omit the WHERE clause; DB2 updates *every row* in the table or view with the values you supply.

If DB2 finds an error while executing your UPDATE statement (for instance, an update value that is too large for the column), it stops updating and returns error codes in the SQLCODE and SQLSTATE host variables or related fields in the SQLCA. No rows in the table change (rows already changed, if any, are restored to their previous values). If the UPDATE statement is successful, SQLERRD(3) is set to the number of rows updated.

Examples: The following statement supplies a missing middle initial and changes the job for employee 000200.

```
UPDATE YEMP
  SET MIDINIT = 'H', JOB = 'FIELDREP'
  WHERE EMPNO = '000200';
```

The following statement gives everyone in department D11 a $400 raise. The statement can update several rows.

```
UPDATE YEMP
  SET SALARY = SALARY + 400.00
  WHERE WORKDEPT = 'D11';
```

The following statement sets the salary and bonus for employee 000190 to the average salary and minimum bonus for all employees.

```
UPDATE YEMP
  SET (SALARY, BONUS) =
  (SELECT AVG(SALARY), MIN(BONUS)
    FROM EMP)
   WHERE EMPNO = '000190';
```

Deleting rows: DELETE

You can use the DELETE statement to remove entire rows from a table. The DELETE statement removes zero or more rows of a table, depending on how many rows satisfy the search condition you specified in the WHERE clause. If you omit a WHERE clause from a DELETE statement, DB2 removes *all the rows* from the table or view you have named. The DELETE statement does not remove specific columns from the row.

You can use DELETE to remove all rows from a created temporary table or declared temporary table. However, you can use DELETE with a WHERE clause to remove selected rows only from a declared temporary table.

This DELETE statement deletes each row in the YEMP table that has an employee number 000060.

```
DELETE FROM YEMP
  WHERE EMPNO = '000060';
```

When this statement executes, DB2 deletes any row from the YEMP table that meets the search condition.

If DB2 finds an error while executing your DELETE statement, it stops deleting data and returns error codes in the SQLCODE and SQLSTATE host variables or related fields in the SQLCA. The data in the table does not change.

If the DELETE is successful, SQLERRD(3) in the SQLCA contains the number of deleted rows. This number includes only the number of rows deleted in the table specified in the DELETE statement. It does not include those rows deleted according to the CASCADE rule.

Deleting every row in a table

The DELETE statement is a powerful statement that deletes *all* rows of a table unless you specify a WHERE clause to limit it. (With segmented table spaces, deleting all rows of a table is very fast.) For example, this statement:

```
DELETE FROM YDEPT;
```

deletes *every row* in the YDEPT table. If the statement executes, the table continues to exist (that is, you can insert rows into it) but it is empty. All existing views and authorizations on the table remain intact when using DELETE. By comparison, using DROP TABLE drops all views and authorizations, which can invalidate plans and packages. For information on the DROP statement, see "Dropping tables: DROP TABLE" on page 23.

Chapter 3. Joining data from more than one table

Sometimes the information you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table. You can use a SELECT statement to retrieve and join column values from two or more tables into a single row.

DB2 supports these types of joins: inner join, left outer join, right outer join, and full outer join.

You can specify joins in the FROM clause of a query: Figure 2 below shows the ways to combine tables using outer join functions.
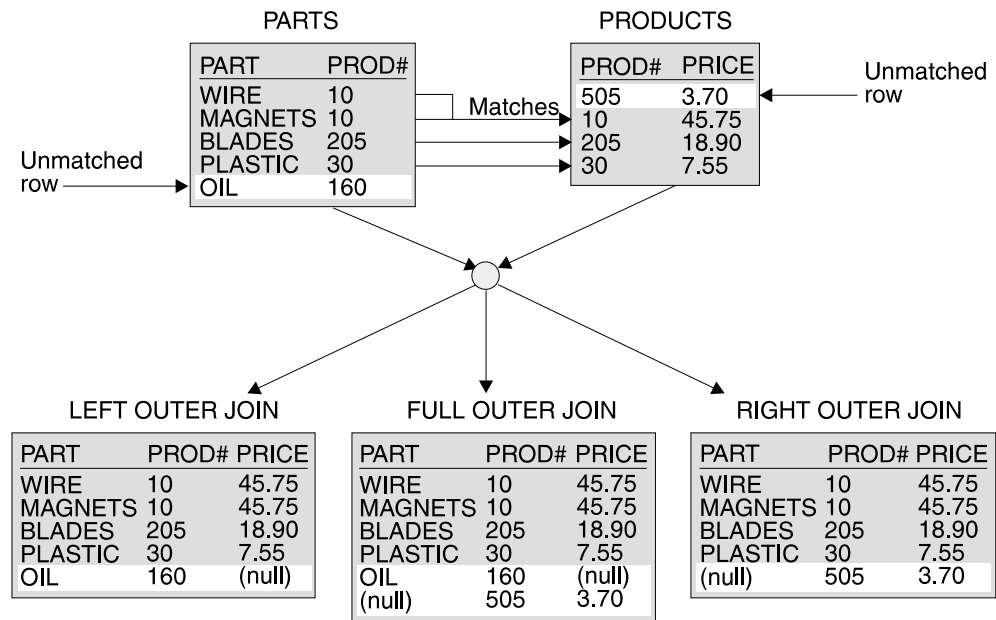
Figure 2. Outer joins of two tables. Each join is on column PROD#.

The result table contains data joined from all of the tables, for rows that satisfy the search conditions.

The result columns of a join have names if the outermost SELECT list refers to base columns. But, if you use a function (such as COALESCE or VALUE) to build a column of the result, then that column does not have a name unless you use the AS clause in the SELECT list.

To distinguish the different types of joins, the examples in this section use the following two tables:

```
The PARTS table                                 The PRODUCTS table
PART       PROD#   SUPPLIER          PROD#    PRODUCT        PRICE
=======    =====   ============      =====    ==========     =====
WIRE       10      ACWF              505      SCREWDRIVER    3.70
OIL        160     WESTERN_CHEM      30       RELAY          7.55
MAGNETS    10      BATEMAN           205      SAW            18.90
PLASTIC    30      PLASTIK_CORP      10       GENERATOR      45.75
BLADES     205     ACE_STEEL
```

Inner join

To request an inner join, execute a SELECT statement in which you specify the tables that you want to join in the FROM clause, and specify a WHERE clause or an ON clause to indicate the join condition. The join condition can be any simple or compound search condition that does not contain a subquery reference. See Chapter 4 of *DB2 SQL Reference* for the complete syntax of a join condition.

In the simplest type of inner join, the join condition condition is *column1=column2*. For example, you can join the PARTS and PRODUCTS tables on the PROD# column to get a table of parts with their suppliers and the products that use the parts.

Either one of these examples:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS, PRODUCTS
  WHERE PARTS.PROD# = PRODUCTS.PROD#;
```

or

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART          SUPPLIER        PROD#     PRODUCT
=======       ============    =====     ==========
WIRE          ACWF            10        GENERATOR
MAGNETS       BATEMAN         10        GENERATOR
PLASTIC       PLASTIK_CORP    30        RELAY
BLADES        ACE_STEEL       205       SAW
```

Notice three things about the example:

- There is a part in the parts table (OIL) whose product (#160) is not in the products table. There is a product (SCREWDRIVER, #505) that has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.

  An *outer join*, however, includes rows where the values in the joined columns do not match.

- There is an explicit syntax to express that this join is not an outer join but an inner join. You can use INNER JOIN in the FROM clause instead of the comma. Use ON to specify the join condition (rather than WHERE) when you explicitly join tables in the FROM clause.

- If you do not specify a WHERE clause in the first form of the query, the result table contains all possible combinations of rows for the tables identified in the FROM clause. You can obtain the same result by specifying a join condition that is always true in the second form of the query. For example:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON 1=1;
```

  In either case, the number of rows in the result table is the product of the number of rows in each table.

You can specify more complicated join conditions to obtain different sets of results. For example, to eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers and products, write a query like this:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    AND SUPPLIER NOT LIKE 'A%';
```

The result of the query is all rows that do not have a supplier that begins with A:

```
PART         SUPPLIER       PROD#    PRODUCT
=======      ============   =====    ==========
MAGNETS      BATEMAN        10       GENERATOR
PLASTIC      PLASTIK_CORP   30       RELAY
```

Example of joining a table to itself using an inner join: The following example
joins table DSN8710.PROJ to itself and returns the number and name of each
"major" project followed by the number and name of the project that is part of it. In
this example, A indicates the first instance of table DSN8710.PROJ and B indicates
a second instance of this table. The join condition is such that the value in column
PROJNO in table DSN8710.PROJ A must be equal to a value in column MAJPROJ
in table DSN8710.PROJ B.

The SQL statement is:

```
SELECT A.PROJNO, A.PROJNAME, B.PROJNO, B.PROJNAME
  FROM DSN8710.PROJ A, DSN8710.PROJ B
  WHERE A.PROJNO = B.MAJPROJ;
```

The result table is:

```
PROJNO  PROJNAME                   PROJNO    PROJNAME
======  ========================   =======   ========================
AD3100  ADMIN SERVICES             AD3110    GENERAL AD SYSTEMS
AD3110  GENERAL AD SYSTEMS         AD3111    PAYROLL PROGRAMMING
AD3110  GENERAL AD SYSTEMS         AD3112    PERSONNEL PROGRAMMG
.
.
.
OP2010  SYSTEMS SUPPORT            OP2013    DB/DC SUPPORT
```

In this example, the comma in the FROM clause implicitly specifies an inner join,
and acts the same as if the INNER JOIN keywords had been used. When you use
the comma for an inner join, you must specify the join condition on the WHERE
clause. When you use the INNER JOIN keywords, you must specify the join
condition on the ON clause.

Full outer join

The clause FULL OUTER JOIN includes unmatched rows from both tables. Missing
values in a row of the result table contain nulls.

The join condition for a full outer join must be a simple search condition that
compares two columns or cast functions that contain columns.

For example, the following query performs a full outer join of the PARTS and
PRODUCTS tables:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table from the query is:

```
PART         SUPPLIER       PROD#    PRODUCT
=======      ============   =====    ==========
WIRE         ACWF           10       GENERATOR
MAGNETS      BATEMAN        10       GENERATOR
```

```
PLASTIC     PLASTIK_CORP  30      RELAY
BLADES      ACE_STEEL     205     SAW
OIL         WESTERN_CHEM  160     -----------
-------     ------------  ---     SCREWDRIVER
```

Example of Using COALESCE (or VALUE): "COALESCE" is the keyword specified by the SQL standard as a synonym for the VALUE function. The function, by either name, can be particularly useful in full outer join operations, because it returns the first nonnull value.

You probably noticed that the result of the example for "Full outer join" on page 35 is null for SCREWDRIVER, even though the PRODUCTS table contains a product number for SCREWDRIVER. If you select PRODUCTS.PROD# instead, PROD# is null for OIL. If you select both PRODUCTS.PROD# and PARTS.PROD#, the result contains two columns, with both columns contain some null values. We can merge data from both columns into a single column, eliminating the null values, using the COALESCE function.

With the same PARTS and PRODUCTS tables, this example:

```
SELECT PART, SUPPLIER,
   COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT
     FROM PARTS FULL OUTER JOIN PRODUCTS
       ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART        SUPPLIER      PRODNUM  PRODUCT
=======     ============  =======  ==========
WIRE        ACWF          10       GENERATOR
MAGNETS     BATEMAN       10       GENERATOR
PLASTIC     PLASTIK_CORP  30       RELAY
BLADES      ACE_STEEL     205      SAW
OIL         WESTERN_CHEM  160      -----------
-------     ------------  505      SCREWDRIVER
```

The AS clause (AS PRODNUM) provides a name for the result of the COALESCE function.

Left outer join

The clause LEFT OUTER JOIN includes rows from the table that is specified before LEFT OUTER JOIN that have no matching values in the table that is specified after LEFT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

For example, to include rows from the PARTS table that have no matching values in the PRODUCTS table and include only prices of greater than 10.00, execute this query:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS LEFT OUTER JOIN PRODUCTS
    ON PARTS.PROD#=PRODUCTS.PROD#
    AND PRODUCTS.PRICE>10.00;
```

The result of the query is:

```
PART        SUPPLIER      PROD#  PRODUCT       PRICE
=======     ============  =====  ==========    =====
WIRE        ACWF          10     GENERATOR     45.75
```

```
MAGNETS      BATEMAN       10      GENERATOR     45.75
PLASTIC      PLASTIK_CORP  30      -----------   -------
BLADES       ACE_STEEL     205     SAW           18.90
OIL          WESTERN_CHEM  160     -----------   -------
```

Because the PARTS table can have nonmatching rows, and the PRICE column is not in the PARTS table, rows in which PRICE is less than 10.00 are included in the result of the join, but PRICE is set to null.

Right outer join

The clause RIGHT OUTER JOIN includes rows from the table that is specified after RIGHT OUTER JOIN that have no matching values in the table that is specified before RIGHT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

For example, to include rows from the PRODUCTS table that have no matching values in the PARTS table and include only prices of greater than 10.00, execute this query:

```
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT
  FROM PARTS RIGHT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    AND PRODUCTS.PRICE>10.00;
```

gives this result:

```
PART         SUPPLIER      PROD#   PRODUCT       PRICE
=======      ============  =====   ==========    =====
WIRE         ACWF          10      GENERATOR     45.75
MAGNETS      BATEMAN       10      GENERATOR     45.75
BLADES       ACE_STEEL     205     SAW           18.90
```

Because the PRODUCTS table cannot have nonmatching rows, and the PRICE column is in the PRODUCTS table, rows in which PRICE is less than 10.00 are not included in the result of the join.

SQL rules for statements containing join operations

SQL rules dictate that the result of a SELECT statement look as if the clauses had been evaluated in this order:
- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT

A join operation is part of a FROM clause; therefore, for the purpose of predicting which rows will be returned from a SELECT statement containing a join operation, assume that the join operation is performed first.

For example, suppose that you want to obtain a list of part names, supplier names, product numbers, and product names from the PARTS and PRODUCTS tables. These categories correspond to the PART, SUPPLIER, PROD#, and PRODUCT columns. You want to include rows from either table where the PROD# value does

not match a PROD# value in the other table, which means that you need to do a full outer join. You also want to exclude rows for product number 10. If you code a SELECT statement like this:

```
SELECT PART, SUPPLIER,
  VALUE(PARTS.PROD#,PRODUCTS.PROD#) AS PRODNUM, PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    WHERE PARTS.PROD# <> '10' AND PRODUCTS.PROD# <> '10';
```

you get this table:

```
PART        SUPPLIER        PRODNUM   PRODUCT
=======     ============    =======   ===========
PLASTIC     PLASTIK_CORP    30        RELAY
BLADES      ACE_STEEL       205       SAW
```

which is not the desired result. DB2 performs the join operation first, then applies the WHERE clause. The WHERE clause excludes rows where PROD# has a null value, so the result is the same as if you had specified an inner join.

A correct SELECT statement to produce the list is:

```
SELECT PART, SUPPLIER,
  VALUE(X.PROD#, Y.PROD#) AS PRODNUM, PRODUCT
  FROM
    (SELECT PART, SUPPLIER, PROD# FROM PARTS WHERE PROD# <> '10') X
    FULL OUTER JOIN
    (SELECT PROD#, PRODUCT FROM PRODUCTS WHERE PROD# <> '10') Y
    ON X.PROD# = Y.PROD#;
```

In this case, DB2 applies the WHERE clause to each table separately, so that no rows are eliminated because PROD# is null. DB2 then performs the full outer join operation, and the desired table is obtained:

```
PART        SUPPLIER        PRODNUM   PRODUCT
=======     ============    =======   ===========
OIL         WESTERN_CHEM    160       -----------
BLADES      ACE_STEEL       205       SAW
PLASTIC     PLASTIK_CORP    30        RELAY
-------     ------------    505       SCREWDRIVER
```

Using more than one type of join in an SQL statement

When you need to join more than two tables, you can use more than one join type in the FROM clause. Suppose you wanted a result table showing all the employees, their department names, and the projects they are responsible for, if any. You would need to join three tables to get all the information. For example, you might use a SELECT statement similar to the following:

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM DSN8710.EMP INNER JOIN DSN8710.DEPT
      ON WORKDEPT = DSN8710.DEPT.DEPTNO
    LEFT OUTER JOIN DSN8710.PROJ
      ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S';
```

The result table is:

```
EMPNO    LASTNAME        DEPTNAME                PROJNO
======   =========       =====================   ======
000020   THOMPSON        PLANNING                PL2100
000060   STERN           MANUFACTURING SYSTEMS   MA2110
000100   SPENSER         SOFTWARE SUPPORT        OP2010
000170   YOSHIMURA       MANUFACTURING SYSTEMS   ------
000180   SCOUTTEN        MANUFACTURING SYSTEMS   ------
```

```
000190  WALKER          MANUFACTURING SYSTEMS   ------
000250  SMITH           ADMINISTRATION SYSTEMS  AD3112
000280  SCHNEIDER       OPERATIONS              ------
000300  SMITH           OPERATIONS              ------
000310  SETRIGHT        OPERATIONS              ------
200170  YAMAMOTO        MANUFACTURING SYSTEMS   ------
200280  SCHWARTZ        OPERATIONS              ------
200310  SPRINGER        OPERATIONS              ------
200330  WONG            SOFTWARE SUPPORT        ------
```

Using nested table expressions and user-defined table functions in joins

An operand of a join can be more complex than the name of a single table. You can use:

- A nested table expression

  A nested table expression is a fullselect enclosed in parentheses, followed by a correlation name.

- A user-defined table function

  A user-defined table function is a user-defined function that returns a table.

The following query contains a nested table expression:

```
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
       PRODUCT, PART, UNITS
  FROM PROJECTS LEFT JOIN
     (SELECT PART,
         COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
         PRODUCTS.PRODUCT
         FROM PARTS FULL OUTER JOIN PRODUCTS
           ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
  ON PROJECTS.PROD# = PRODNUM;
```

The nested table expression is this:

```
(SELECT PART,
    COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
    PRODUCTS.PRODUCT
    FROM PARTS FULL OUTER JOIN PRODUCTS
       ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
```

The correlation name is TEMP.

Example of using a simple nested table expression:

```
SELECT CHEAP_PARTS.PROD#, CHEAP_PARTS.PRODUCT
  FROM (SELECT PROD#, PRODUCT
            FROM PRODUCTS
               WHERE PRICE < 10) AS CHEAP_PARTS;
```

gives this result:

```
PROD#     PRODUCT
=====     ===========
505       SCREWDRIVER
30        RELAY
```

In the example, the correlation name is CHEAP_PARTS. There are two correlated references to CHEAP_PARTS: CHEAP_PARTS.PROD# and CHEAP_PARTS.PRODUCT. Those references are valid because they do not occur in the same FROM clause where CHEAP_PARTS is defined.

Example of a fullselect as the left operand of a join:

```
SELECT PART, SUPPLIER, PRODNUM, PRODUCT
  FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
              FROM PARTS
                  WHERE PROD# < '200') AS PARTX
          LEFT OUTER JOIN PRODUCTS
              ON PRODNUM = PROD#;
```

gives this result:

```
PART         SUPPLIER       PRODNUM      PRODUCT
=======      ============   =======      ==========
WIRE         ACWF           10           GENERATOR
MAGNETS      BATEMAN        10           GENERATOR
OIL          WESTERN_CHEM   160          ----------
```

Because PROD# is a character field, DB2 does a character comparison to determine the set of rows in the result. Therefore, because '30' is greater than ssq;200', the row in which PROD# is equal to '30' does not appear in the result.

Example of a join with a table function:

You can join the results of a user-defined table function with a table, just as you can join two tables. For example, suppose CVTPRICE is a table function that converts the prices in the PRODUCTS table to the currency you specify and returns the PRODUCTS table with the prices in those units. You can obtain a table of parts, suppliers, and product prices with the prices in your choice of currency by executing a query like this:

```
SELECT PART, SUPPLIER, PARTS.PROD#, Z.PRODUCT, Z.PRICE
  FROM PARTS, TABLE(CVTPRICE(:CURRENCY)) AS Z
  WHERE PARTS.PROD# = Z.PROD#;
```

Examples of correlated references in table references:

You can include correlated references in nested table expressions or as arguments to table functions. The basic rule that applies for both these cases is that the correlated reference must be from a table specification at a higher level in the hierarchy of subqueries. You can also use a correlated reference and the table specification to which it refers in the same FROM clause if the table specification appears to the left of the correlated reference and the correlated reference is in one of the following clauses:

- A nested table expression preceded by the keyword TABLE
- The argument of a table function

A table function or a table expression that contains correlated references to other tables in the same FROM clause cannot participate in a full outer join or a right outer join.

The following examples illustrate valid uses of correlated references in table specifications:

```
SELECT T.C1, Z.C5
  FROM T, TABLE(TF3(T.C2)) AS Z
  WHERE T.C3 = Z.C4;
```

The correlated reference T.C2 is valid because the table specification, to which it refers, T, is to its left. If you specify the join in the opposite order, with T following TABLE(TF3(T.C2), then T.C2 is invalid.

```
SELECT D.DEPTNO, D.DEPTNAME,
  EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
  FROM DEPT D,
    TABLE(SELECT AVG(E.SALARY) AS AVGSAL,
          COUNT(*) AS EMPCOUNT
          FROM EMP E
          WHERE E.WORKDEPT=D.DEPTNO) AS EMPINFO;
```

The correlated reference D.DEPTNO is valid because the nested table expression within which it appears is preceded by TABLE and the table specification D appears to the left of the nested table expression in the FROM clause. If you remove the keyword TABLE, D.DEPTNO is invalid.

Chapter 4. Using subqueries

You should use a subquery when you need to narrow your search condition based on information in an interim table. For example, you might want to find all employee numbers in one table that also exist for a given project in a second table.

This chapter presents a conceptual overview of subqueries, shows how to include subqueries in either a WHERE or a HAVING clause, and shows how to use correlated subqueries.

Conceptual overview

Suppose you want a list of the employee numbers, names, and commissions of all employees working on a particular project, say project number MA2111. The first part of the SELECT statement is easy to write:

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8710.EMP
  WHERE EMPNO
       .
       .
       .
```

But you cannot go further because the DSN8710.EMP table does not include project number data. You do not know which employees are working on project MA2111 without issuing another SELECT statement against the DSN8710.EMPPROJACT table.

You can use a *subquery* to solve this problem. A *subquery* is a subselect or a fullselect in a WHERE clause. The SELECT statement surrounding the subquery is called the *outer SELECT*.

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8710.EMP
  WHERE EMPNO IN
    (SELECT EMPNO
       FROM DSN8710.EMPPROJACT
        WHERE PROJNO = 'MA2111');
```

To better understand what results from this SQL statement, imagine that DB2 goes through the following process:

1. DB2 evaluates the subquery to obtain a list of EMPNO values:

   ```
   (SELECT EMPNO
       FROM DSN8710.EMPPROJACT
       WHERE PROJNO = 'MA2111');
   ```

   which results in an *interim result table*:

   (From EMPPROJACT)

   ```
   000200
   000200
   000220
   ```

2. The interim result table then serves as a list in the search condition of the outer SELECT. Effectively, DB2 executes this statement:

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8710.EMP
  WHERE EMPNO IN
  ('000200', '000220');
```

As a consequence, the result table looks like this:

	EMPNO	LASTNAME	COMM
Fetch			
1 →	000200	BROWN	2217
2 →	000220	LUTZ	2387

Correlated and uncorrelated subqueries

Subqueries supply information needed to qualify a row (in a WHERE clause) or a group of rows (in a HAVING clause). The subquery produces a result table used to qualify the row or group of rows selected. The subquery executes only once, if the subquery is the same for every row or group.

This kind of subquery is *uncorrelated*. In the previous query, for example, the content of the subquery is the same for every row of the table DSN8710.EMP.

Subqueries that vary in content from row to row or group to group are *correlated* subqueries. For information on correlated subqueries, see "Using correlated subqueries" on page 47. All of the information preceding that section applies to both correlated and uncorrelated subqueries.

Subqueries and predicates

A subquery is always part of a predicate. The predicate is of the form:

operand operator (subquery)

The predicate can be part of a WHERE or HAVING clause. A WHERE or HAVING clause can include predicates that contain subqueries. A predicate containing a subquery, like any other search predicate, can be enclosed in parentheses, can be preceded by the keyword NOT, and can be linked to other predicates through the keywords AND and OR. For example, the WHERE clause of a query could look something like this:

WHERE X IN (*subquery1*) AND (Y > SOME (*subquery2*) OR Z IS NULL)

Subqueries can also appear in the predicates of other subqueries. Such subqueries are *nested* subqueries at some *level of nesting*. For example, a subquery within a subquery within an outer SELECT has a level of nesting of 2. DB2 allows nesting down to a level of 15, but few queries require a nesting level greater than 1.

The relationship of a subquery to its outer SELECT is the same as the relationship of a nested subquery to a subquery, and the same rules apply, except where otherwise noted.

The subquery result table

A subquery must produce a result table that has the same number of columns as the number of columns on the left side of the comparison operator. For example, both of the following SELECT statements are acceptable:

```
SELECT EMPNO, LASTNAME
  FROM DSN8710.EMP
  WHERE SALARY=
  (SELECT AVG(SALARY)
    FROM DSN8710.EMP);
SELECT EMPNO, LASTNAME
  FROM DSN8710.EMP
  WHERE (SALARY, BONUS) IN
  (SELECT AVG(SALARY), AVG(BONUS)
    FROM DSN8710.EMP);
```

Except for a subquery of a basic predicate, the result table can contain more than one row.

Tables in subqueries of UPDATE, DELETE, and INSERT statements

The following rules apply to a table that is used in a subquery for an UPDATE, DELETE, or INSERT statement:

- When you use a subquery in an INSERT statement, the subquery can use the same table as the INSERT statement.

- When you use a subquery in a *searched* UPDATE or DELETE statement (an UPDATE or DELETE that does not use a cursor), the subquery can use the same table as the UPDATE or DELETE statement.

- When you use a subquery in a *positioned* UPDATE or DELETE statement (an UPDATE or DELETE that uses a cursor), the subquery cannot use the same table as the UPDATE or DELETE statement.

How to code a subquery

There are a number of ways to specify a subquery in either a WHERE or HAVING clause. They are as follows:
- Basic predicate
- Quantified Predicates: ALL, ANY, and SOME
- Using the IN Keyword
- Using the EXISTS Keyword

Basic predicate

You can use a subquery immediately after any of the comparison operators. If you do, the subquery can return at most one value. DB2 compares that value with the value to the left of the comparison operator.

For example, the following SQL statement returns the employee numbers, names, and salaries for employees whose education level is higher than the average company-wide education level.

```
SELECT EMPNO, LASTNAME, SALARY
  FROM DSN8710.EMP
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
      FROM DSN8710.EMP);
```

Quantified predicates: ALL, ANY, and SOME

You can use a subquery after a comparison operator followed by the keyword ALL, ANY, or SOME. The number of columns and rows that the subquery can return for a quantified predicate depends on the type of quantified predicate:

- For = SOME, = ANY, or <> ALL, the subquery can return one or many rows and one or many columns. The number of columns in the result table must match the number of columns on the left side of the operator.
- For all other quantified predicates, the subquery can return one or many rows, but no more than one column.

Use ALL to indicate that the operands on the left side of the comparison must compare in the same way with *all* the values the subquery returns. For example, suppose you use the greater-than comparison operator with ALL:

```
WHERE column > ALL (subquery)
```

To satisfy this WHERE clause, the column value must be greater than all the values that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Now suppose that you use the <> operator with ALL in a WHERE clause like this:

```
WHERE column1, column1, ... columnn <> ALL (subquery)
```

To satisfy this WHERE clause, each column value must be unequal to all the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Use ANY or SOME to indicate that the values on the left side of the operator must compare in the indicated way to *at least one* of the values that the subquery returns. For example, suppose you use the greater-than comparison operator with ANY:

```
WHERE expression > ANY (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (that is, greater than the lowest value) that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

Now suppose that you use the = operator with SOME in a WHERE clause like this:

```
WHERE column1, column1, ... columnn = SOME (subquery)
```

To satisfy this WHERE clause, each column value must be equal to at least one of the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

If a subquery that returns one or more null values gives you unexpected results, see the description of quantified predicates in Chapter 2 of *DB2 SQL Reference*.

Using the IN keyword

You can use IN to say that the value or values on the left side of the IN operator must be among the values that are returned by the subquery. Using IN is equivalent to using = ANY or = SOME.

Using the EXISTS keyword

In the subqueries presented thus far, DB2 evaluates the subquery and uses the result as part of the WHERE clause of the outer SELECT. In contrast, when you use the keyword EXISTS, DB2 simply checks whether the subquery returns one or more rows. Returning one or more rows satisfies the condition; returning no rows does not satisfy the condition. For example:

```
SELECT EMPNO,LASTNAME
  FROM DSN8710.EMP
    WHERE EXISTS
      (SELECT *
         FROM DSN8710.PROJ
          WHERE PRSTDATE > '1986-01-01');
```

In the example, the search condition is true if any project represented in the DSN8710.PROJ table has an estimated start date which is later than 1 January 1986. This example does not show the full power of EXISTS, because the result is always the same for every row examined for the outer SELECT. As a consequence, either every row appears in the results, or none appear. A correlated subquery is more powerful, because the subquery would change from row to row.

As shown in the example, you do not need to specify column names in the subquery of an EXISTS clause. Instead, you can code `SELECT *`. You can also use the EXISTS keyword with the NOT keyword in order to select rows when the data or condition you specify *does not* exist; that is, you can code

`WHERE NOT EXISTS (SELECT ...);`

Using correlated subqueries

In the subqueries previously described, DB2 executes the subquery once, substitutes the result of the subquery in the right side of the search condition, and evaluates the outer-level SELECT based on the value of the search condition. You can also write a subquery that DB2 has to re-evaluate when it examines a new row (in a WHERE clause) or group of rows (in a HAVING clause) as it executes the outer SELECT. This is called a *correlated subquery*.

User-defined functions in correlated subqueries: Use care when you invoke a user-defined function in a correlated subquery, and that user-defined function uses a scratchpad. DB2 does not refresh the scratchpad between invocations of the subquery. This can cause undesirable results because the scratchpad keeps values across the invocations of the subquery.

An example of a correlated subquery

Suppose that you want a list of all the employees whose education levels are higher than the average education levels in their respective departments. To get this information, DB2 must search the DSN8710.EMP table. For each employee in the table, DB2 needs to compare the employee's education level to the average education level for the employee's department.

This is the point at which a correlated subquery differs from an uncorrelated subquery. The earlier example of uncorrelated subqueries compares the education level to the average of the entire company, which requires looking at the entire table. A correlated subquery evaluates only the department which corresponds to the particular employee.

In the subquery, you tell DB2 to compute the average education level for the department number in the current row. A query that does this follows:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
  FROM DSN8710.EMP X
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM DSN8710.EMP
        WHERE WORKDEPT = X.WORKDEPT);
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more *correlated references*. In the example, the single correlated reference is the occurrence of X.WORKDEPT in the WHERE clause of the subselect. In this clause, the qualifier X is the *correlation name* defined in the FROM clause of the outer SELECT statement. X designates rows of the first instance of DSN8710.EMP. At any time during the execution of the query, X designates the row of DSN8710.EMP to which the WHERE clause is being applied.

Consider what happens when the subquery executes for a given row of DSN8710.EMP. Before it executes, X.WORKDEPT receives the value of the WORKDEPT column for that row. Suppose, for example, that the row is for CHRISTINE HAAS. Her work department is A00, which is the value of WORKDEPT for that row. The subquery executed for that row is therefore:

```
(SELECT AVG(EDLEVEL)
    FROM DSN8710.EMP
    WHERE WORKDEPT = 'A00');
```

The subquery produces the average education level of Christine's department. The outer subselect then compares this to Christine's own education level. For some other row for which WORKDEPT has a different value, that value appears in the subquery in place of A00. For example, in the row for MICHAEL L THOMPSON, this value is B01, and the subquery for his row delivers the average education level for department B01.

The result table produced by the query has the following values:

(From EMP)

Fetch	EMPNO	LASTNAME	WORKDEPT	EDLEVEL
1 →	000010	HAAS	A00	18
2 →	000030	KWAN	C01	20
3 →	000070	PULASKI	D21	16
4 →	000090	HENDERSON	E11	16

Using correlation names in references

A correlated reference can appear in a subquery, in a nested table expression, or as an argument of a user-defined table function. For information on correlated references in nested table expressions and table functions, see "Using nested table expressions and user-defined table functions in joins" on page 39. In a subquery, the reference should be of the form X.C, where X is a correlation name and C is the name of a column in the table that X represents.

Any number of correlated references can appear in a subquery. There are no restrictions on variety. For example, you can define one correlated name in a reference in the outer SELECT, and another in a nested subquery.

When you use a correlation name in a subquery, the subquery can be the outer-level SELECT, or any of the subqueries that contain the reference. Suppose, for example, that a query contains subqueries A, B, and C, and that A contains B and B contains C. Then C could use a correlation name defined in B, A, or the outer SELECT.

You can define a correlation name for each table name appearing in a FROM clause. Append the correlation name after its table name. Leave one or more blanks between a table name and its correlation name. You can include the word AS between the table name and the correlation name to increase the readability of

the SQL statement. The following example demonstrates the use of a correlation name in the search condition of a subquery:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
  FROM DSN8710.EMP AS X
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM DSN8710.EMP
       WHERE WORKDEPT = X.WORKDEPT);
```

The following example demonstrates the use of a correlation name in the select list of a subquery:

```
UPDATE BP1TBL T1
  SET (KEY1, CHAR1, VCHAR1) =
    (SELECT VALUE(T2.KEY1,T1.KEY1), VALUE(T2.CHAR1,T1.CHAR1), VALUE(T2.VCHAR1,T1.VCHAR1)
       FROM BP2TBL T2
       WHERE (T2.KEY1 = T1.KEY1))
  WHERE KEY1 IN
  (SELECT KEY1
    FROM BP2TBL T3
    WHERE KEY2 > 0);
```

Using correlated subqueries in an UPDATE statement

When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows you are updating. For example, when all activities of a project must complete before September 1997, your department considers that project to be a priority project. You can use the following SQL statement to evaluate the projects in the DSN8710.PROJ table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column (a column you have added to DSN8710.PROJ for this purpose) for each priority project:

```
UPDATE DSN8710.PROJ X
SET PRIORITY = 1
WHERE DATE('1997-09-01') >
  (SELECT MAX(ACENDATE)
     FROM DSN8710.PROJACT
     WHERE PROJNO = X.PROJNO);
```

As DB2 examines each row in the DSN8710.PROJ table, it determines the maximum activity end date (ACENDATE) for all activities of the project (from the DSN8710.PROJACT table). If the end date of each activity associated with the project is before September 1997, the current row in the DSN8710.PROJ table qualifies and DB2 updates it.

Using correlated subqueries in a DELETE statement

When you use a correlated subquery in a DELETE statement, the correlation name represents the row you delete. DB2 evaluates the correlated subquery once for each row in the table named in the DELETE statement to decide whether or not to delete the row.

For example, suppose that a department considers a project to be complete when the combined amount of time currently spent on it is half a person's time or less. The department then deletes the rows for that project from the DSN8710.PROJ table. In the example statements that follow, PROJ and PROJACT are independent tables; that is, they are separate tables with no referential constraints defined on them.

```
DELETE FROM DSN8710.PROJ X
  WHERE .5 >
    (SELECT SUM(ACSTAFF)
       FROM DSN8710.PROJACT
       WHERE PROJNO = X.PROJNO);
```

To process this statement, DB2 determines for each project (represented by a row in the DSN8710.PROJ table) whether or not the combined staffing for that project is less than 0.5. If it is, DB2 deletes that row from the DSN8710.PROJ table.

To continue this example, suppose DB2 deletes a row in the DSN8710.PROJ table. You must also delete rows related to the deleted project in the DSN8710.PROJACT table. To do this, use:

```
DELETE FROM DSN8710.PROJACT X
  WHERE NOT EXISTS
    (SELECT *
       FROM DSN8710.PROJ
       WHERE PROJNO = X.PROJNO);
```

DB2 determines, for each row in the DSN8710.PROJACT table, whether a row with the same project number exists in the DSN8710.PROJ table. If not, DB2 deletes the row in DSN8710.PROJACT.

A subquery of a searched DELETE statement (a DELETE statement that does not use a cursor) can reference the same table from which rows are deleted. In the following statement, which deletes the employee with the highest salary from each department, the employee table appears in the outer DELETE and in the subselect:

```
DELETE FROM YEMP X
  WHERE SALARY =(SELECT MAX(SALARY) FROM YEMP Y
                   WHERE X.WORKDEPT =Y.WORKDEPT);
```

This example uses copies of the employee and department table that do not have referential constraints.

DB2 restricts delete operations for dependent tables that are involved in referential constraints. If a DELETE statement has a subquery that references a table involved in the deletion, the last delete rule in the path to that table must be RESTRICT or NO ACTION. For example, without referential constraints, the following statement deletes departments from the department table whose managers are not listed correctly in the employee table:

```
DELETE FROM DSN8710.DEPT THIS
  WHERE NOT DEPTNO =
    (SELECT WORKDEPT
       FROM DSN8710.EMP
       WHERE EMPNO = THIS.MGRNO);
```

With the referential constraints defined for the sample tables, the statement causes an error. The deletion involves the table referred to in the subquery (DSN8710.EMP is a dependent table of DSN8710.DEPT) and the last delete rule in the path to EMP is SET NULL, not RESTRICT or NO ACTION. If the statement could execute, its results would again depend on the order in which DB2 accesses the rows.

Chapter 5. Executing SQL from your terminal using SPUFI

This chapter explains how to enter and execute SQL statements at a TSO terminal using the SPUFI (SQL processor using file input) facility. You can execute most of the interactive SQL examples shown in Part 1. Using SQL queries by following the instructions provided in this chapter and using the sample tables shown in "Appendix A. DB2 sample tables" on page 811. The instructions assume that ISPF is available to you.

Allocating an input data set and using SPUFI

Before you use SPUFI, you should allocate an input data set, if one does not already exist. This data set will contain one or more SQL statements that you want to execute. For information on ISPF and allocating data sets, refer to *ISPF V4 User's Guide*.

To use SPUFI, select SPUFI from the DB2I Primary Option Menu as shown in Figure 3.

```
DSNEPRI                    DB2I PRIMARY OPTION MENU          SSID: DSN
COMMAND ===> 1

Select one of the following DB2 functions and press ENTER.

 1  SPUFI                  (Process SQL statements)
 2  DCLGEN                 (Generate SQL and source language declarations)
 3  PROGRAM PREPARATION    (Prepare a DB2 application program to run)
 4  PRECOMPILE             (Invoke DB2 precompiler)
 5  BIND/REBIND/FREE       (BIND, REBIND, or FREE plans or packages)
 6  RUN                    (RUN an SQL program)
 7  DB2 COMMANDS           (Issue DB2 commands)
 8  UTILITIES              (Invoke DB2 utilities)
 D  DB2I DEFAULTS          (Set global parameters)
 X  EXIT                   (Leave DB2I)



PRESS:  END to exit        HELP for more information
```

Figure 3. The DB2I primary option menu with option 1 selected

The SPUFI panel then displays as shown in Figure 4 on page 52.

From then on, when the SPUFI panel displays, the data entry fields on the panel contain the values that you previously entered. You can specify data set names and processing options each time the SPUFI panel displays, as needed. Values you do not change remain in effect.

```
 DSNESP01                         SPUFI                      SSID: DSN
 ===>
 Enter the input data set name:   (Can be sequential or partitioned)
  1 DATA SET NAME..... ===> EXAMPLES(XMP1)
  2 VOLUME SERIAL..... ===>        (Enter if not cataloged)
  3 DATA SET PASSWORD. ===>        (Enter if password protected)

Enter the output data set name:   (Must be a sequential data set)
  4 DATA SET NAME..... ===> RESULT


Specify processing options:
  5 CHANGE DEFAULTS... ===> Y      (Y/N - Display SPUFI defaults panel?)
  6 EDIT INPUT........ ===> Y      (Y/N - Enter SQL statements?)
  7 EXECUTE........... ===> Y      (Y/N - Execute SQL statements?)
  8 AUTOCOMMIT........ ===> Y      (Y/N - Commit after successful run?)
  9 BROWSE OUTPUT..... ===> Y      (Y/N - Browse output data set?)

 For remote SQL processing:
 10 CONNECT LOCATION   ===>


 PRESS: ENTER to process      END to exit     HELP for more information
```

Figure 4. The SPUFI panel filled in

Fill out the SPUFI panel as follows:

1,2,3 INPUT DATA SET NAME
> Identify the input data set in fields 1 through 3. This data set contains one or more SQL statements that you want to execute. Allocate this data set before you use SPUFI, if one does not already exist.
>
> - The name must conform to standard TSO naming conventions.
> - The data set can be empty before you begin the session. You can then add the SQL statements by editing the data set from SPUFI.
> - The data set can be either sequential or partitioned, but it must have the following DCB characteristics:
>   - A record format (RECFM) of either F or FB.
>   - A logical record length (LRECL) of either 79 or 80. Use 80 for any data set that the EXPORT command of QMF did not create.
> - Data in the data set can begin in column 1. It can extend to column 71 if the logical record length is 79, and to column 72 if the logical record length is 80. SPUFI assumes that the last 8 bytes of each record are for sequence numbers.
>
> If you use this panel a second time, the name of the data set you previously used displays in the field DATA SET NAME. To create a new member of an existing partitioned data set, change only the member name.

4 OUTPUT DATA SET NAME
> Enter the name of a data set to receive the output of the SQL statement. You do not need to allocate the data set before you do this.
>
> If the data set exists, the new output replaces its content. If the data set does not exist, DB2 allocates a data set on the device type specified on the CURRENT SPUFI DEFAULTS panel and then catalogs the new data set. The device must be a direct-access storage device, and you must be authorized to allocate space on that device.
>
> Attributes required for the output data set are:

- Organization: sequential
- Record format: F, FB, FBA, V, VB, or VBA
- Record length: 80 to 32768 bytes, not less than the input data set

Figure 4 on page 52 shows the simplest choice, entering **RESULT**. SPUFI allocates a data set named *userid*.RESULT and sends all output to that data set. If a data set named *userid*.RESULT already exists, SPUFI sends DB2 output to it, replacing all existing data.

5 CHANGE DEFAULTS

Allows you to change control values and characteristics of the output data set and format of your SPUFI session. If you specify Y(YES) you can look at the SPUFI defaults panel. See "Changing SPUFI defaults (optional)" on page 54 for more information about the values you can specify and how they affect SPUFI processing and output characteristics. You do not need to change the SPUFI defaults for this example.

6 EDIT INPUT

To edit the input data set, leave Y(YES) on line 6. You can use the ISPF editor to create a new member of the input data set and enter SQL statements in it. (To process a data set that already contains a set of SQL statements you want to execute immediately, enter **N**(NO). Specifying N bypasses the step described in "Entering SQL statements" on page 56.)

7 EXECUTE

To execute SQL statements contained in the input data set, leave Y(YES) on line 7.

SPUFI handles the SQL statements that can be dynamically prepared. For those SQL statements, see "Appendix G. Characteristics of SQL statements in DB2 for OS/390 and z/OS" on page 919.

8 AUTOCOMMIT

To make changes to the DB2 data permanent, leave Y(YES) on line 8. Specifying Y makes SPUFI issue COMMIT if all statements execute successfully. If all statements do not execute successfully, SPUFI issues a ROLLBACK statement, which deletes changes already made to the file (back to the last commit point). Please read about the COMMIT and the ROLLBACK functions in "Unit of work in TSO (batch and online)" on page 357 or Chapter 5 of *DB2 SQL Reference*.

If you specify N, DB2 displays the SPUFI COMMIT OR ROLLBACK panel after it executes the SQL in your input data set. That panel prompts you to COMMIT, ROLLBACK, or DEFER any updates made by the SQL. If you enter DEFER, you neither commit nor roll back your changes.

9 BROWSE OUTPUT

To look at the results of your query, leave Y(YES) on line 9. SPUFI saves the results in the output data set. You can look at them at any time, until you delete or write over the data set. For more information, see "Format of SELECT statement results" on page 58.

10 CONNECT LOCATION

Specify the name of the database server, if applicable, to which you want to submit SQL statements. SPUFI then issues a type 2 CONNECT statement to this server.

SPUFI is a locally bound package. SQL statements in the input data set can process only if the CONNECT statement is successful. If the connect request fails, the output data set contains the resulting SQL return codes and error messages.

Changing SPUFI defaults (optional)

When you finish with the SPUFI panel, press the ENTER key. Because you specified YES on line 5 of the SPUFI panel, the next panel you see is the SPUFI Defaults panel. SPUFI provides default values the first time you use SPUFI, for all options except the DB2 subsystem name. Any changes you make to these values remain in effect until you change the values again. Figure 5 shows the initial default values.

```
DSNESP02                  CURRENT SPUFI DEFAULTS                 SSID: DSN
===>
Enter the following to control your SPUFI session:
 1  SQL TERMINATOR     ===> ;       (SQL Statement Terminator)
 2  ISOLATION LEVEL    ===> RR      (RR=Repeatable Read, CS=Cursor Stability)
 3  MAX SELECT LINES   ===> 250     (Maximum number of lines to be
                                       returned from a SELECT)
Output data set characteristics:
 4  RECORD LENGTH ... ===> 4092     (LRECL= logical record length)
 5  BLOCKSIZE ....... ===> 4096     (Size of one block)
 6  RECORD FORMAT.... ===> VB       (RECFM= F, FB, FBA, V, VB, or VB)
 7  DEVICE TYPE...... ===> SYSDA    (Must be a DASD unit name)


Output format characteristics:
 8  MAX NUMERIC FIELD ===> 33       (Maximum width for numeric field)
 9  MAX CHAR FIELD .. ===> 80       (Maximum width for character field)
10  COLUMN HEADING .. ===> NAMES    (NAMES, LABELS, ANY, or BOTH)




PRESS: ENTER to process     END to exit    HELP for more information
```

Figure 5. The SPUFI defaults panel

Specify values for the following options on the CURRENT SPUFI DEFAULTS panel. All fields must contain a value.

1 SQL TERMINATOR

Allows you to specify the character that you use to end each SQL statement. You can specify any character *except* one of those listed in Table 3. A semicolon is the default.

Table 3. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal Representation
blank		X'40'
comma	,	X'5E'
double quote	"	X'7F'
left parenthesis	(	X'4D'
right parenthesis	)	X'5D'
single quote	'	X'7D'
underscore	_	X'6D'

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose you choose the character # as the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

Be careful to choose a character for the SQL terminator that is not used within the statement.

You can also set or change the SQL terminator within a SPUFI input data set using the `--#SET TERMINATOR` statement. See "Entering SQL statements" on page 56 for details.

2 ISOLATION LEVEL

Allows you to specify the isolation level for your SQL statements. See "The ISOLATION option" on page 341 for more information.

3 MAX SELECT LINES

The maximum number of output lines that a SELECT statement can return. To limit the number of rows retrieved, enter another maximum number greater than 1.

4 RECORD LENGTH

The record length must be at least 80 bytes. The maximum record length depends on the device type you use. The default value allows a 4092-byte record.

Each record can hold a single line of output. If a line is longer than a record, the last fields in the line truncate. SPUFI discards fields beyond the record length.

5 BLOCKSIZE

Follow the normal rules for selecting the block size. For record format F, the block size is equal to record length. For FB and FBA, choose a block size that is an even multiple of LRECL. For VB and VBA only, the block size must be 4 bytes larger than the block size for FB or FBA.

6 RECORD FORMAT

Specify F, FB, FBA, V, VB, or VBA. FBA and VBA formats insert a printer control character after the number of lines specified in the LINES/PAGE OF LISTING field on the DB2I Defaults panel. The record format default is VB (variable-length blocked).

7 DEVICE TYPE

Allows you to specify a standard MVS name for direct-access storage device types. The default is SYSDA. SYSDA specifies that MVS is to select an appropriate direct access storage device.

8 MAX NUMERIC FIELD

The maximum width of a numeric value column in your output. Choose a value greater than 0. The IBM-supplied default is 20. For more information, see "Format of SELECT statement results" on page 58.

9 MAX CHAR FIELD

The maximum width of a character value column in your output. DATETIME and GRAPHIC data strings are externally represented as characters, and

SPUFI includes their defaults with the default values for character fields. Choose a value greater than 0. The IBM-supplied default is 80. For more information, see "Format of SELECT statement results" on page 58.

10 COLUMN HEADING

You can specify NAMES, LABELS, ANY or BOTH for column headings.
- NAME (default) uses column names only.
- LABEL uses column labels. Leave the title blank if there is no label.
- ANY uses existing column labels or column names.
- BOTH creates two title lines, one with names and one with labels.

Column names are the column identifiers that you can use in SQL statements. If an SQL statement has an AS clause for a column, SPUFI displays the contents of the AS clause in the heading, rather than the column name. You define column labels with LABEL ON statements.

When you have entered your SPUFI options, press the ENTER key to continue. SPUFI then processes the next processing option for which you specified YES. If all other processing options are NO, SPUFI displays the SPUFI panel.

If you press the END key, you return to the SPUFI panel, but you lose all the changes you made on the SPUFI Defaults panel. If you press ENTER, SPUFI saves your changes.

Entering SQL statements

Next, SPUFI lets you edit the input data set. Initially, editing consists of entering an SQL statement into the input data set. You can also edit an input data set that contains SQL statements and you can change, delete, or insert SQL statements.

The ISPF Editor shows you an empty EDIT panel.

On the panel, use the ISPF EDIT program to enter SQL statements that you want to execute, as shown in Figure 6 on page 57.

Move the cursor to the first input line and enter the first part of an SQL statement. You can enter the rest of the SQL statement on subsequent lines, as shown in Figure 6 on page 57. Indenting your lines and entering your statements on several lines make your statements easier to read, and do not change how your statements process.

You can put more than one SQL statement in the input data set. You can put an SQL statement on one line of the input data set or on more than one line. DB2 executes the statements in the order you placed them in the data set. Do not put more than one SQL statement on a single line. The first one executes, but DB2 ignores the other SQL statements on the same line.

In your SPUFI input data set, end each SQL statement with the statement terminator that you specified in the CURRENT SPUFI DEFAULTS panel.

When you have entered your SQL statements, press the END PF key to save the file and to execute the SQL statements.

```
EDIT --------userid.EXAMPLES(XMP1) --------------------- COLUMNS 001 072
COMMAND INPUT ===> SAVE                            SCROLL ===> PAGE
******************************** TOP OF DATA ***********************
000100 SELECT LASTNAME, FIRSTNME, PHONENO
000200    FROM DSN8710.EMP
000300    WHERE WORKDEPT= 'D11'
000400    ORDER BY LASTNAME;
****************************** BOTTOM OF DATA *********************
```

Figure 6. The edit panel: After entering an SQL statement

Pressing the END PF key saves the data set. You can save the data set *and* continue editing it by entering the SAVE command. In fact, it is a good practice to save the data set after every 10 minutes or so of editing.

Figure 6 shows what the panel looks like if you enter the sample SQL statement, followed by a SAVE command.

You can bypass the editing step by resetting the EDIT INPUT processing option:
EDIT INPUT ... ===> NO

You can put comments about SQL statements either on separate lines or on the same line. In either case, use two hyphens (--) to begin a comment. Specify any text other than #SET TERMINATOR after the comment. DB2 ignores everything to the right of the two hyphens.

Use the text --SET TERMINATOR *character* in a SPUFI input data set as an instruction to SPUFI to interpret *character* as a statement terminator. You can specify any single-byte character *except* one of the characters that are listed in Table 3 on page 54. The terminator that you specify overrides a terminator that you specified in option 1 of the CURRENT SPUFI DEFAULTS panel or in a previous --SET TERMINATOR statement.

Processing SQL statements

SPUFI passes the input data set to DB2 for processing. DB2 executes the SQL statement in the input data set EXAMPLES(XMP1), and sends the output to the output data set *userid*.RESULT.

You can bypass the DB2 processing step by resetting the EXECUTE processing option:
EXECUTE ..... ===> NO

Your SQL statement might take a long time to execute, depending on how large a table DB2 has to search, or on how many rows DB2 has to process. To interrupt DB2's processing, press the PA1 key and respond to the prompting message that asks you if you really want to stop processing. This cancels the executing SQL statement and returns you to the ISPF-PDF menu.

What happens to the output data set? This depends on how much of the input data set DB2 was able to process before you interrupted its processing. DB2 might not have opened the output data set yet, or the output data set might contain all or part of the results data produced so far.

Browsing the output

SPUFI formats and displays the output data set using the ISPF Browse program. Figure 7 shows the output from the sample program. An output data set contains these items for each SQL statement that DB2 executes:

- The executed SQL statement, copied from the input data set
- The results of executing the SQL statement
- The formatted SQLCA, if an error occurs during statement execution

At the end of the data set are summary statistics that describe the processing of the input data set as a whole.

When executing a SELECT statement using SPUFI, the message "SQLCODE IS 100" indicates an error-free result. If the message SQLCODE IS 100 is the only result, DB2 is unable to find any rows that satisfy the condition specified in the statement.

For all other types of SQL statements executed with SPUFI, the message "SQLCODE IS 0" indicates an error-free result.

```
  BROWSE-- userid.RESULT                          COLUMNS 001 072
  COMMAND INPUT ===>                                       SCROLL ===> PAGE
  --------+---------+---------+---------+---------+---------+---------+---------+
  SELECT LASTNAME, FIRSTNME, PHONENO                             00010000
   FROM DSN8710.EMP                                              00020000
   WHERE WORKDEPT = 'D11'                                        00030000
   ORDER BY LASTNAME;                                            00040000
  ---------+---------+---------+---------+---------+---------+---------+---------+
  LASTNAME        FIRSTNME     PHONENO
  ADAMSON         BRUCE        4510
  BROWN           DAVID        4501
  JOHN            REBA         0672
  JONES           WILLIAM      0942
  LUTZ            JENNIFER     0672
  PIANKA          ELIZABETH    3782
  SCOUTTEN        MARILYN      1682
  STERN           IRVING       6423
  WALKER          JAMES        2986
  YAMAMOTO        KIYOSHI      2890
  YOSHIMURA       MASATOSHI    2890
  DSNE610I NUMBER OF ROWS DISPLAYED IS 11
  DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
  ---------+---------+---------+---------+---------+---------+---------+----
  ---------+---------+---------+---------+---------+---------+---------+----
  DSNE617I COMMIT PERFORMED, SQLCODE IS 0
  DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
  ---------+---------+---------+---------+---------+---------+---------+----
  DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
  DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
  DSNE621I NUMBER OF INPUT RECORDS READ IS 4
  DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 30
```

Figure 7. Result data set from the sample problem

Format of SELECT statement results

The results of SELECT statements follow these rules:

- If a column's numeric or character data cannot display completely:
  - Character values that are too wide truncate on the right.
  - Numeric values that are too wide display as asterisks (*).

- For columns other than LOB columns, if truncation occurs, the output data set contains a warning message. Because LOB columns are generally longer than the value you choose for field MAX CHAR FIELD on panel CURRENT SPUFI DEFAULTS, SPUFI displays no warning message when it truncates LOB column output.

  You can change the amount of data displayed for numeric and character columns by changing values on the CURRENT SPUFI DEFAULTS panel, as described in "Changing SPUFI defaults (optional)" on page 54.
- A null value displays as a series of hyphens (-).
- A ROWID or BLOB column value displays in hexadecimal.
- A CLOB column value displays in the same way as a VARCHAR column value.
- A DBCLOB column value displays in the same way as a VARGRAPHIC column value.
- A heading identifies each selected column, and repeats at the top of each output page. The contents of the heading depend on the value you specified in field COLUMN HEADING of the CURRENT SPUFI DEFAULTS panel.

Content of the messages

Each message contains the following:
- The SQLCODE, if the statement executes successfully
- The formatted SQLCA, if the statement executes unsuccessfully
- What character positions of the input data set that SPUFI scanned to find SQL statements. This information helps you check the assumptions SPUFI made about the location of line numbers (if any) in your input data set.
- Some overall statistics:
  - Number of SQL statements processed
  - Number of input records read (from the input data set)
  - Number of output records written (to the output data set).

Other messages that you could receive from the processing of SQL statements include:
- The number of rows that DB2 processed, that either:
  - Your SELECT statement retrieved
  - Your UPDATE statement modified
  - Your INSERT statement added to a table
  - Your DELETE statement deleted from a table
- Which columns display truncated data because the data was too wide

Part 2. Coding SQL in your host application program

Chapter 6. Basics of coding SQL in an application program

Suppose you are writing an application program to access data in a DB2 database. When your program executes an SQL statement, the program needs to communicate with DB2. When DB2 finishes processing an SQL statement, DB2 sends back a return code, and your program should test the return code to examine the results of the operation.

To communicate with DB2, you need to:
- Choose a method for communicating with DB2. You can use one of these methods:
  - Static SQL
  - Embedded dynamic SQL
  - Open Database Connectivity (ODBC)
  - JDBC application support

  This book discusses embedded SQL. See "Chapter 23. Coding dynamic SQL in application programs" on page 495 for a comparison of static and embedded dynamic SQL and an extended discussion of embedded dynamic SQL.

  ODBC lets you access data through ODBC function calls in your application. You execute SQL statements by passing them to DB2 through a ODBC function call. ODBC eliminates the need for precompiling and binding your application and increases the portability of your application by using the ODBC interface.

  If you are writing your applications in Java, you can use JDBC application support to access DB2. JDBC is similar to ODBC but is designed specifically for use with Java and is therefore a better choice than ODBC for making DB2 calls from Java applications.

  For more information on using JDBC, see *DB2 ODBC Guide and Reference*.
- Delimit SQL statements, as described in "Delimiting an SQL statement" on page 66.
- Declare the tables you use, as described in "Declaring table and view definitions" on page 67. (This is optional.)
- Declare the data items used to pass data between DB2 and a host language, as described in "Accessing data using host variables and host structures" on page 67.
- Code SQL statements to access DB2 data. See "Accessing data using host variables and host structures" on page 67.

  For information about using the SQL language, see "Part 1. Using SQL queries" on page 1 and in *DB2 SQL Reference*. Details about how to use SQL statements within an application program are described in "Chapter 9. Embedding SQL statements in host languages" on page 107.
- Declare a communications area (SQLCA), or handle exceptional conditions that DB2 indicates with return codes, in the SQLCA. See "Checking the execution of SQL statements" on page 74 for more information.

In addition to these basic requirements, you should also consider several special topics:
- "Chapter 7. Using a cursor to retrieve a set of rows" on page 81 discusses how to use a cursor in your application program to select a set of rows and then process the set one row at a time.

- "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95 discusses how to use DB2's declarations generator, DCLGEN, to obtain accurate SQL DECLARE statements for tables and views.

This section includes information about using SQL in application programs written in assembler, C, COBOL, FORTRAN, PL/I, and REXX. You can also use SQL in application programs written in Ada, APL2®, BASIC, and Prolog. See the following publications for more information about these languages:

Ada   *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide*

APL2   *APL2 Programming: Using Structured Query Language (SQL)*

BASIC   *IBM BASIC/MVS Language Reference*

Prolog/MVS & VM
          IBM SAA AD/Cycle® Prolog/MVS & VM Programmer's Guide

Conventions used in examples of coding SQL statements

The SQL statements shown in this section use the following conventions:

- The SQL statement is part of a COBOL application program. Each SQL example shows on several lines, with each clause of the statement on a separate line.
- The use of the precompiler options APOST and APOSTSQL are assumed (although they are not the defaults). Hence, apostrophes (') are used to delimit character string literals within SQL and host language statements.
- The SQL statements access data in the sample tables provided with DB2. The tables contain data that a manufacturing company might keep about its employees and its current projects. For a description of the tables, see "Appendix A. DB2 sample tables" on page 811.
- An SQL example does not necessarily show the complete syntax of an SQL statement. For the complete description and syntax of any of the statements described in this book, see Chapter 5 of *DB2 SQL Reference*.
- Examples do not take referential constraints into account. For more information about how referential constraints affect SQL statements, and examples of how SQL statements operate with referential constraints, see "Chapter 2. Working with tables and modifying data" on page 17.

Some of the examples vary from these conventions. Exceptions are noted where they occur.

Delimiting an SQL statement

For languages other than REXX, bracket an SQL statement in your program between `EXEC SQL` and a statement terminator. The terminators for the languages described in this book are:

Language	SQL Statement Terminator
Assembler	End of line or end of last continued line
C	Semicolon (;)
COBOL	`END-EXEC`
FORTRAN	End of line or end of last continued line
PL/I	Semicolon (;)

For REXX, precede the statement with EXECSQL. If the statement is in a literal string, enclose it in single or double quotation marks.

For example, use EXEC SQL and END-EXEC to delimit an SQL statement in a COBOL program:

```
EXEC SQL
   an SQL statement
END-EXEC.
```

Declaring table and view definitions

Before your program issues SQL statements that retrieve, update, delete, or insert data, you should declare the tables and views your program accesses. To do this, include an SQL DECLARE statement in your program.

You do not have to declare tables or views, but there are advantages if you do. One advantage is documentation. For example, the DECLARE statement specifies the structure of the table or view you are working with, and the data type of each column. You can refer to the DECLARE statement for the column names and data types in the table or view. Another advantage is that the DB2 precompiler uses your declarations to make sure you have used correct column names and data types in your SQL statements. The DB2 precompiler issues a warning message when the column names and data types do not correspond to the SQL DECLARE statements in your program.

A way to declare a table or view is to code a DECLARE statement in the WORKING-STORAGE SECTION or LINKAGE SECTION within the DATA DIVISION of your COBOL program. Specify the name of the table and list each column and its data type. When you declare a table or view, you specify DECLARE *table-name* TABLE regardless of whether the table-name refers to a table or a view.

For example, the DECLARE TABLE statement for the DSN8710.DEPT table looks like this:

```
EXEC SQL
 DECLARE DSN8710.DEPT TABLE
    (DEPTNO    CHAR(3)          NOT NULL,
     DEPTNAME  VARCHAR(36)      NOT NULL,
     MGRNO     CHAR(6)                  ,
     ADMRDEPT  CHAR(3)          NOT NULL,
     LOCATION  CHAR(16)                 )

END-EXEC.
```

As an alternative to coding the DECLARE statement yourself, you can use DCLGEN, the declarations generator supplied with DB2. For more information about using DCLGEN, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

When you declare a table or view that contains a column with a distinct type, it is best to declare that column with the source type of the distinct type, rather than the distinct type itself. When you declare the column with the source type, DB2 can check embedded SQL statements that reference that column at precompile time.

Accessing data using host variables and host structures

You can access data using host variables and host structures.

A *host variable* is a data item declared in the host language for use within an SQL statement. Using host variables, you can:

- Retrieve data into the host variable for your application program's use
- Place data into the host variable to insert into a table or to change the contents of a row
- Use the data in the host variable when evaluating a WHERE or HAVING clause
- Assign the value in the host variable to a special register, such as CURRENT SQLID and CURRENT DEGREE
- Insert null values in columns using a host indicator variable that contains a negative value
- Use the data in the host variable in statements that process dynamic SQL, such as EXECUTE, PREPARE, and OPEN

A *host structure* is a group of host variables that an SQL statement can refer to using a single name. You can use host structures in all languages except REXX. Use host language statements to define the host structures.

Using host variables

You can use any valid host variable name in an SQL statement. You must declare the name in the host program before you use it. (For more information see the appropriate language section in "Chapter 9. Embedding SQL statements in host languages" on page 107.)

To optimize performance, make sure the host language declaration maps as closely as possible to the data type of the associated data in the database; see "Chapter 9. Embedding SQL statements in host languages" on page 107. For more performance suggestions, see "Part 6. Additional programming techniques" on page 487.

You can use a host variable to represent a data value, but you cannot use it to represent a table, view, or column name. (You can specify table, view, or column names at run time using dynamic SQL. See "Chapter 23. Coding dynamic SQL in application programs" on page 495 for more information.)

Host variables follow the naming conventions of the host language. A colon (:) must precede host variables used in SQL to tell DB2 that the variable is not a column name. A colon must *not* precede host variables outside of SQL statements.

For more information about declaring host variables, see the appropriate language section:

- *Assembler*: "Using host variables" on page 111
- *C*: "Using host variables" on page 124
- *COBOL*: "Using host variables" on page 146
- *FORTRAN*: "Using host variables" on page 167
- *PL/I*: "Using host variables" on page 177.
- *REXX*: "Using REXX host variables and data types" on page 194.

Retrieving data into a host variable
You can use a host variable to specify a program data area to contain the column values of a retrieved row or rows.

Retrieving a single row of data: The INTO clause of the SELECT statement names one or more host variables to contain the column values returned. The named variables correspond one-to-one with the list of column names in the SELECT list.

For example, suppose you are retrieving the EMPNO, LASTNAME, and WORKDEPT column values from rows in the DSN8710.EMP table. You can define a data area in your program to hold each column, then name the data areas with an INTO clause, as in the following example. (Notice that a colon precedes each host variable):

```
EXEC SQL
  SELECT EMPNO, LASTNAME, WORKDEPT
    INTO :CBLEMPNO, :CBLNAME, :CBLDEPT
    FROM DSN8710.EMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

In the DATA DIVISION of the program, you must declare the host variables CBLEMPNO, CBLNAME, and CBLDEPT to be compatible with the data types in the columns EMPNO, LASTNAME, and WORKDEPT of the DSN8710.EMP table.

If the SELECT statement returns more than one row, this is an error, and any data returned is undefined and unpredictable.

Retrieving Multiple Rows of Data: If you do not know how many rows DB2 will return, or if you expect more than one row to return, then you must use an alternative to the SELECT ... INTO statement.

The DB2 *cursor* enables an application to process a set of rows and retrieve one row at a time from the result table. For information on using cursors, see "Chapter 7. Using a cursor to retrieve a set of rows" on page 81.

Specifying a list of items in a select clause: When you specify a list of items in the SELECT clause, you can use more than the column names of tables and views. You can request a set of column values mixed with host variable values and constants. For example:

```
MOVE 4476 TO RAISE.
MOVE '000220' TO PERSON.
EXEC SQL
    SELECT EMPNO, LASTNAME, SALARY, :RAISE, SALARY + :RAISE
      INTO :EMP-NUM, :PERSON-NAME, :EMP-SAL, :EMP-RAISE, :EMP-TTL
    FROM DSN8710.EMP
    WHERE EMPNO = :PERSON
END-EXEC.
```

The results shown below have column headings that represent the names of the host variables:

```
EMP-NUM    PERSON-NAME    EMP-SAL    EMP-RAISE    EMP-TTL
=======    ===========    =======    =========    =======
 000220    LUTZ             29840         4476      34316
```

Inserting and updating data

You can set or change a value in a DB2 table to the value of a host variable. To do this, you can use the host variable name in the SET clause of UPDATE or the VALUES clause of INSERT. This example changes an employee's phone number:

```
EXEC SQL
  UPDATE DSN8710.EMP
    SET PHONENO = :NEWPHONE
    WHERE EMPNO = :EMPID
END-EXEC.
```

Searching data

You can use a host variable to specify a value in the predicate of a search condition or to replace a constant in an expression. For example, if you have defined a field called EMPID that contains an employee number, you can retrieve the name of the employee whose number is 000110 with:

```
MOVE '000110' TO EMPID.
EXEC SQL
  SELECT LASTNAME
    INTO :PGM-LASTNAME
    FROM DSN8710.EMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

Using indicator variables with host variables

Indicator variables are small integers that you can use to:
- Determine whether the value of an associated output host variable is null or indicate that an input host variable value is null
- Determine the original length of a character string that was truncated during assignment to a host variable
- Determine that a character value could not be converted during assignment to a host variable
- Determine the seconds portion of a time value that was truncated during assignment to a host variable

Retrieving data into host variables: If the value for the column you retrieve is null, DB2 puts a negative value in the indicator variable. If it is null because of a numeric or character conversion error, or an arithmetic expression error, DB2 sets the indicator variable to -2. See "Handling arithmetic or conversion errors" on page 75 for more information.

If you do not use an indicator variable and DB2 retrieves a null value, an error results.

When DB2 retrieves the value of a column, you can test the indicator variable. If the indicator variable's value is less than zero, the column value is null. When the column value is null, the value of the host variable does not change from its previous value.

You can also use an indicator variable to verify that a retrieved character string value is not truncated. If the indicator variable contains a positive integer, the integer is the original length of the string.

You can specify an indicator variable, preceded by a colon, immediately after the host variable. Optionally, you can use the word INDICATOR between the host variable and its indicator variable. Thus, the following two examples are equivalent:

```
EXEC SQL                              EXEC SQL
  SELECT PHONENO                        SELECT PHONENO
    INTO :CBLPHONE:INDNULL                INTO :CBLPHONE INDICATOR :INDNULL
    FROM DSN8710.EMP                      FROM DSN8710.EMP
    WHERE EMPNO = :EMPID                  WHERE EMPNO = :EMPID
END-EXEC.                             END-EXEC.
```

You can then test INDNULL for a negative value. If it is negative, the corresponding value of PHONENO is null, and you can disregard the contents of CBLPHONE.

When you use a cursor to fetch a column value, you can use the same technique to determine whether the column value is null.

Inserting null values into columns using host variables: You can use an indicator variable to insert a null value from a host variable into a column. When DB2 processes INSERT and UPDATE statements, it checks the indicator variable (if it exists). If the indicator variable is negative, the column value is null. If the indicator variable is greater than -1, the associated host variable contains a value for the column.

For example, suppose your program reads an employee ID and a new phone number, and must update the employee table with the new number. The new number could be missing if the old number is incorrect, but a new number is not yet available. If it is possible that the new value for column PHONENO might be null, you can code:

```
EXEC SQL
  UPDATE DSN8710.EMP
    SET PHONENO = :NEWPHONE:PHONEIND
    WHERE EMPNO = :EMPID
END-EXEC.
```

When NEWPHONE contains other than a null value, set PHONEIND to zero by preceding the statement with:

```
MOVE 0 TO PHONEIND.
```

When NEWPHONE contains a null value, set PHONEIND to a negative value by preceding the statement with:

```
MOVE -1 TO PHONEIND.
```

Use IS NULL to test for a null column value: You cannot determine whether a column value is null by comparing a host variable with an indicator variable that is set -1 to the column. Two DB2 null values are not equal to each other. To test whether a column has a null value, use the IS NULL comparison operator. For example, the following code does *not* select the employees who do no have a phone number:

```
    MOVE -1 TO PHONE-IND.
    EXEC SQL
      SELECT LASTNAME
        INTO :PGM-LASTNAME
        FROM DSN8710.EMP
        WHERE PHONENO = :PHONE-HV:PHONE-IND
    END-EXEC.
```

To obtain that information, use a statement like this one:

```
    EXEC SQL
      SELECT LASTNAME
        INTO :PGM-LASTNAME
        FROM DSN8710.EMP
        WHERE PHONENO IS NULL
    END-EXEC.
```

Assignments and comparisons using different data types

For assignments and comparisons involving a DB2 column and a host variable of a different data type or length, you can expect conversions to occur. If you assign or compare data, see Chapter 2 of *DB2 SQL Reference* for the rules associated with these operations.

Changing the coded character set ID of host variables

All DB2 string data, other than BLOB data, has an encoding scheme and a coded character set ID (CCSID) associated with it. You can use the DECLARE VARIABLE statement to associate an encoding scheme and a CCSID with individual host variables. The DECLARE VARIABLE statement has the following effects on a host variable:

- When you use the host variable to update a table, the local subsystem or the remote server assumes that the data in the host variable is encoded with the CCSID and encoding scheme that the DECLARE VARIABLE statement assigns.

- When you retrieve data from a local or remote table into the host variable, the retrieved data is converted to the CCSID and encoding scheme that are assigned by the DECLARE VARIABLE statement.

You can use the DECLARE VARIABLE statement in static or dynamic SQL applications. However, you cannot use the DECLARE VARIABLE statement to control the CCSID and encoding scheme of data that you retrieve or update using an SQLDA. See "Changing the CCSID for retrieved data" on page 517 for information on changing the CCSID in an SQLDA.

When you use a DECLARE VARIABLE statement for a host variable in a program, you need to put the DECLARE VARIABLE statement after the host variable declaration and before you refer to that host variable.

Example: Using a DECLARE VARIABLE statement to change the encoding scheme of retrieved data: Suppose that you are writing a C program that runs on a DB2 for OS/390 and z/OS system that has an application encoding scheme of EBCDIC. The C program retrieves data from the following columns of a local table that is defined with CCSID UNICODE.

```
PARTNUM CHAR(10)
JPNNAME GRAPHIC(10)
ENGNAME VARCHAR(30)
```

Because the application encoding scheme for the subsystem is EBCDIC, the retrieved data is EBCDIC. To make the retrieved data Unicode, use DECLARE VARIABLE statements to specify that the data that is retrieved from these columns is encoded in the default Unicode CCSIDs for the subsystem. Suppose that you want to retrieve the character data in Unicode CCSID 1208 and the graphic data in Unicode CCSID 1200. Use DECLARE VARIABLE statements like these:

```
EXEC SQL BEGIN DECLARE SECTION;
char hvpartnum[11]
EXEC SQL DECLARE :hvpartnum VARIABLE CCSID 1208;
wchar_t hvjpnname[11];
EXEC SQL DECLARE :hvjpnname VARIABLE CCSID 1200;
struct {
        short len;
        char d[30];
      } hvengname;
EXEC SQL DECLARE :hvengname VARIABLE CCSID 1208;
EXEC SQL END DECLARE SECTION;
```

Using host structures

You can substitute a host structure for one or more host variables. You can also use indicator variables (or structures) with host structures.

Example: Using a host structure

In the following example, assume that your COBOL program includes the following SQL statement:

```
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
    INTO :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, :WORKDEPT
    FROM DSN8710.VEMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

If you want to avoid listing host variables, you can substitute the name of a structure, say :PEMP, that contains :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, and :WORKDEPT. The example then reads:

```
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
    INTO :PEMP
    FROM DSN8710.VEMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

You can declare a host structure yourself, or you can use DCLGEN to generate a COBOL record description, PL/I structure declaration, or C structure declaration that corresponds to the columns of a table. For more details about coding a host structure in your program, see "Chapter 9. Embedding SQL statements in host languages" on page 107. For more information on using DCLGEN and the restrictions that apply to the C language, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

Using indicator variables with host structures

You can define an *indicator structure* (an array of halfword integer variables) to support a host structure. You define indicator structures in the DATA DIVISION of your COBOL program. If the column values your program retrieves into a host structure can be null, you can attach an indicator structure name to the host structure name. This allows DB2 to notify your program about each null value returned to a host variable in the host structure. For example:

```
01 PEMP-ROW.
   10 EMPNO              PIC X(6).
   10 FIRSTNME.
      49 FIRSTNME-LEN    PIC S9(4) USAGE COMP.
      49 FIRSTNME-TEXT   PIC X(12).
   10 MIDINIT            PIC X(1).
   10 LASTNAME.
      49 LASTNAME-LEN    PIC S9(4) USAGE COMP.
      49 LASTNAME-TEXT   PIC X(15).
   10 WORKDEPT           PIC X(3).
   10 EMP-BIRTHDATE      PIC X(10).
01 INDICATOR-TABLE.
   02 EMP-IND            PIC S9(4) COMP OCCURS 6 TIMES.
   :
   :
MOVE '000230' TO EMPNO.
   :
   :
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, BIRTHDATE
    INTO :PEMP-ROW:EMP-IND
    FROM DSN8710.EMP
    WHERE EMPNO = :EMPNO
END-EXEC.
```

In this example, EMP-IND is an array containing six values, which you can test for negative values. If, for example, EMP-IND(6) contains a negative value, the corresponding host variable in the host structure (EMP-BIRTHDATE) contains a null value.

Because this example selects rows from the table DSN8710.EMP, some of the values in EMP-IND are always zero. The first four columns of each row are defined NOT NULL. In the above example, DB2 selects the values for a row of data into a host structure. You must use a corresponding structure for the indicator variables to determine which (if any) selected column values are null. For information on using the IS NULL keyword phrase in WHERE clauses, see "Chapter 1. Retrieving data" on page 3.

Checking the execution of SQL statements

A program that includes SQL statements needs to have an area set apart for communication with DB2 — an *SQL communication area* (SQLCA). When DB2 processes an SQL statement in your program, it places return codes in the SQLCODE and SQLSTATE host variables or corresponding fields of the SQLCA. The return codes indicate whether the statement executed succeeded or failed.

Because the SQLCA is a valuable problem-diagnosis tool, it is a good idea to include the instructions necessary to display some of the information contained in the SQLCA in your application programs. For example, the contents of SQLERRD(3)—which indicates the number of rows that DB2 updates, inserts, or deletes—could be useful. If SQLWARN0 contains W, DB2 has set at least one of the SQL warning flags (SQLWARN1 through SQLWARNA). See Appendix C of *DB2 SQL Reference* for a description of all the fields in the SQLCA.

SQLCODE and SQLSTATE

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code. Although both fields serve basically the same purpose (indicating whether the statement executed successfully) there are some differences between the two fields.

SQLCODE: DB2 returns the following codes in SQLCODE:
- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

SQLCODE 100 indicates no data was found.

The meaning of SQLCODEs other than 0 and 100 varies with the particular product implementing SQL.

SQLSTATE: SQLSTATE allows an application program to check for errors in the same way for different IBM database management systems. See Appendix C of *DB2 Messages and Codes* for a complete list of possible SQLSTATE values.

An advantage to using the SQLCODE field is that it can provide more specific information than the SQLSTATE. Many of the SQLCODEs have associated tokens in the SQLCA that indicate, for example, which object incurred an SQL error.

To conform to the SQL standard, you can declare SQLCODE and SQLSTATE (SQLCOD and SQLSTA in FORTRAN) as stand-alone host variables. If you specify the STDSQL(YES) precompiler option, these host variables receive the return codes, and you should not include an SQLCA in your program.

The WHENEVER statement

The WHENEVER statement causes DB2 to check the SQLCA and continue processing your program, or branch to another area in your program if an error, exception, or warning exists as a result of executing an SQL statement. Your program can then examine SQLCODE or SQLSTATE to react specifically to the error or exception.

The WHENEVER statement is not supported for REXX. For information on REXX error handling, see "Embedding SQL statements in a REXX procedure" on page 192.

The WHENEVER statement allows you to specify what to do if a general condition is true. You can specify more than one WHENEVER statement in your program. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until the next WHENEVER statement.

The WHENEVER statement looks like this:

```
EXEC SQL
  WHENEVER condition action
END-EXEC
```

Condition is one of these three values:

SQLWARNING
Indicates what to do when SQLWARN0 = W or SQLCODE contains a positive value other than 100. SQLWARN0 can be set for several different reasons — for example, if a column value truncates when it moves into a host variable. It is possible your program would not regard this as an error.

SQLERROR
Indicates what to do when DB2 returns an error code as the result of an SQL statement (SQLCODE < 0).

NOT FOUND
Indicates what to do when DB2 cannot find a row to satisfy your SQL statement or when there are no more rows to fetch (SQLCODE = 100).

Action is one of these two values:

CONTINUE
Specifies the next sequential statement of the source program.

GOTO or GO TO *host-label*
Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, preceded by a colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

The WHENEVER statement *must precede* the first SQL statement it is to affect. However, if your program checks SQLCODE directly, it must check SQLCODE after the SQL statement executes.

Handling arithmetic or conversion errors

Numeric or character conversion errors or arithmetic expression errors can set an indicator variable to -2. For example, division by zero and arithmetic overflow does not necessarily halt the execution of a SELECT statement. If the error occurs in the SELECT list, the statement can continue to execute and return good data for rows in which the error does not occur, if you use indicator variables.

For rows in which the error does occur, one or more selected items have no meaningful value. The indicator variable flags this error with a -2 for the affected host variable, and an SQLCODE of +802 (SQLSTATE '01519') in the SQLCA.

Handling SQL error return codes

You should check for errors before you commit data, and handle the errors that they represent. The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message based on the SQLCODE field of the SQLCA.

You can find the programming language specific syntax and details for calling DSNTIAR on the following pages:

DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. Each time you use DSNTIAR, it overwrites any previous messages in the message output area. You should move or print the messages before using DSNTIAR again, and before the contents of the SQLCA change, to get an accurate view of the SQLCA.

DSNTIAR expects the SQLCA to be in a certain format. If your application modifies the SQLCA format before you call DSNTIAR, the results are unpredictable.

Defining a message output area

The calling program must allocate enough storage in the message output area to hold all of the message text. You will probably not need more than 10 lines of 80 bytes each for your message output area. Your application program can have only one message output area.

You must define the message output area in VARCHAR format. In this varying character format, a two-byte length field precedes the data. The length field tells DSNTIAR how many total bytes are in the output message area; its minimum value is 240.

Figure 8 on page 77 shows the format of the message output area, where *length* is the two-byte total length field, and the length of each line matches the logical record length (*lrecl*) you specify to DSNTIAR.

Line:

Field sizes (in bytes):

2 — Logical record length
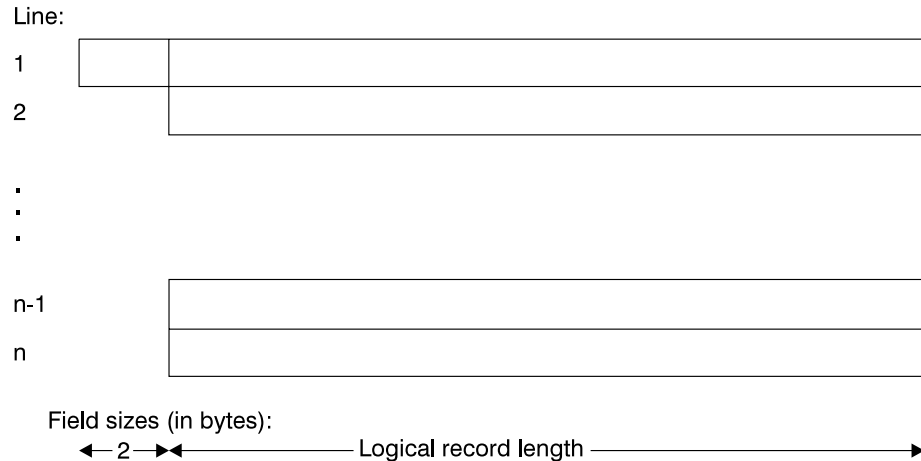
Figure 8. Format of the message output area

When you call DSNTIAR, you must name an SQLCA and an output message area in its parameters. You must also provide the logical record length (*lrecl*) as a value between 72 and 240 bytes. DSNTIAR assumes the message area contains fixed-length records of length *lrecl*.

DSNTIAR places up to 10 lines in the message area. If the text of a message is longer than the record length you specify on DSNTIAR, the output message splits into several records, on word boundaries if possible. The split records are indented. All records begin with a blank character for carriage control. If you have more lines than the message output area can contain, DSNTIAR issues a return code of 4. A completely blank record marks the end of the message output area.

Possible return codes from DSNTIAR

Code	Meaning
0	Successful execution.
4	More data was available than could fit into the provided message area.
8	The logical record length was not between 72 and 240, inclusive.
12	The message area was not large enough. The message length was 240 or greater.
16	Error in TSO message routine.
20	module DSNTIA1 could not be loaded.
24	SQLCA data error.

Preparing to use DSNTIAR

DSNTIAR can run either above or below the 16MB line of virtual storage. The DSNTIAR object module that comes with DB2 has the attributes AMODE(31) and RMODE(ANY). At install time, DSNTIAR links as AMODE(31) and RMODE(ANY). Thus, DSNTIAR runs in 31-bit mode if:

- Linked with other modules that also have the attributes AMODE(31) and RMODE(ANY),
- Linked into an application that specifies the attributes AMODE(31) and RMODE(ANY) in its link-edit JCL, or
- An application loads it.

When loading DSNTIAR from another program, be careful how you branch to DSNTIAR. For example, if the calling program is in 24-bit addressing mode and DSNTIAR is loaded above the 16-megabyte line, you cannot use the assembler BALR instruction or CALL macro to call DSNTIAR, because they assume that

DSNTIAR is in 24-bit mode. Instead, you must use an instruction that is capable of branching into 31-bit mode, such as BASSM.

You can dynamically link (load) and call DSNTIAR directly from a language that does not handle 31-bit addressing (OS/VS COBOL, for example). To do this, link a second version of DSNTIAR with the attributes AMODE(24) and RMODE(24) into another load module library. Or you can write an intermediate assembler language program and that calls DSNTIAR in 31-bit mode; then call that intermediate program in 24-bit mode from your application.

For more information on the allowed and default AMODE and RMODE settings for a particular language, see the application programming guide for that language. For details on how the attributes AMODE and RMODE of an application are determined, see the linkage editor and loader user's guide for the language in which you have written the application.

A scenario for using DSNTIAR

Suppose you want your DB2 COBOL application to check for deadlocks and timeouts, and you want to make sure your cursors are closed before continuing. You use the statement WHENEVER SQLERROR to transfer control to an error routine when your application receives a negative SQLCODE.

In your error routine, you write a section that checks for SQLCODE -911 or -913. You can receive either of these SQLCODEs when there is a deadlock or timeout. When one of these errors occurs, the error routine closes your cursors by issuing the statement:

```
EXEC SQL CLOSE cursor-name
```

An SQLCODE of 0 or -501 from that statement indicates that the close was successful.

You can use DSNTIAR in the error routine to generate the complete message text associated with the negative SQLCODEs.

1. Choose a logical record length (*lrecl*) of the output lines. For this example, assume *lrecl* is 72, to fit on a terminal screen, and is stored in the variable named ERROR-TEXT-LEN.

2. Define a message area in your COBOL application. Assuming you want an area for up to 10 lines of length 72, you should define an area of 720 bytes, plus a 2-byte area that specifies the length of the message output area.

```
01  ERROR-MESSAGE.
      02  ERROR-LEN   PIC S9(4)  COMP VALUE +720.
      02  ERROR-TEXT  PIC X(72)  OCCURS 10 TIMES
                                 INDEXED BY ERROR-INDEX.
77  ERROR-TEXT-LEN      PIC S9(9)  COMP VALUE +72.
```

   For this example, the name of the message area is ERROR-MESSAGE.

3. Make sure you have an SQLCA. For this example, assume the name of the SQLCA is SQLCA.

To display the contents of the SQLCA when SQLCODE is 0 or -501, you should first format the message by calling DSNTIAR after the SQL statement that produces SQLCODE 0 or -501:

```
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

You can then print the message output area just as you would any other variable. Your message might look like the following:

```
DSNT408I SQLCODE = -501, ERROR:  THE CURSOR IDENTIFIED IN A FETCH OR
         CLOSE STATEMENT IS NOT OPEN
DSNT418I SQLSTATE  = 24501 SQLSTATE RETURN CODE
DSNT415I SQLERRP   = DSNXERT SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD   = -315  0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD   = X'FFFFFEC5'  X'00000000'  X'00000000'
         X'FFFFFFFF'  X'00000000'  X'00000000' SQL DIAGNOSTIC
         INFORMATION
```

Chapter 7. Using a cursor to retrieve a set of rows

Use a *cursor* in an application program to retrieve rows from a table or from a result set that is returned by a stored procedure. This chapter explains how your application program can use a cursor to retrieve rows from a table. For information on using a cursor to retrieve rows from a result set, see "Chapter 24. Using stored procedures for client/server processing" on page 525.

How to use a cursor

When you execute a SELECT statement, you retrieve a set of rows. That set of rows is called the *result table*. In an application program, you need a way to retrieve one row at a time from the result table into host variables. A cursor performs that function.

The basic steps in using a cursor are:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See "Step 1: Declare the cursor".
2. Execute an OPEN CURSOR to make the cursor available to the application. See "Step 2: Open the cursor" on page 83.
3. Specify what the program does when all rows have been retrieved. See "Step 3: Specify what to do at end-of-data" on page 83.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See "Step 4: Execute SQL statements" on page 83.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. "Step 5: Close the cursor" on page 85.

Your program can have several cursors, each of which performs the previous steps.

Step 1: Declare the cursor

To define and identify a set of rows to be accessed with a cursor, issue a DECLARE CURSOR statement. The DECLARE CURSOR statement names a cursor and specifies a SELECT statement. The SELECT statement defines the criteria for the rows that will make up the result table. See Chapter 4 of *DB2 SQL Reference* for a complete list of clauses that you can use in the SELECT statement.

The following example shows a simple form of the DECLARE CURSOR statement:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8710.EMP
  END-EXEC.
```

You can use this cursor to list select information on employees.

More complicated cursors might include WHERE clauses or joins of several tables. For example, suppose that you want to use a cursor to list employees who work on a certain project. Declare a cursor like this to identify those employees:

```
EXEC SQL
  DECLARE C2 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8710.EMP X
      WHERE EXISTS
```

```
                    (SELECT *
                       FROM DSN8710.PROJ Y
                       WHERE X.EMPNO=Y.RESPEMP
                       AND Y.PROJNO=:GOODPROJ);
```

Updating a column: You can update columns in the rows that you retrieve.
Updating a row after you use a cursor to retrieve it is called a *positioned* update. If
you intend to perform any positioned updates on the identified table, include the
FOR UPDATE clause. The FOR UPDATE clause has two forms. The first form is
FOR UPDATE OF *column-list.* Use this form when you know in advance which
columns you need to update. The second form of the FOR UPDATE clause is FOR
UPDATE, with no column list. Use this form when you might use the cursor to
update any of the columns of the table.

For example, you can use this cursor to update only the SALARY column of the
employee table:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8710.EMP X
      WHERE EXISTS
        (SELECT *
           FROM DSN8710.PROJ Y
           WHERE X.EMPNO=Y.RESPEMP
           AND Y.PROJNO=:GOODPROJ)
    FOR UPDATE OF SALARY;
```

If you might use the cursor to update any column of the employee table, define the
cursor like this:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8710.EMP X
      WHERE EXISTS
        (SELECT *
           FROM DSN8710.PROJ Y
           WHERE X.EMPNO=Y.RESPEMP
           AND Y.PROJNO=:GOODPROJ)
    FOR UPDATE;
```

DB2 must do more processing when you use the FOR UPDATE clause without a
column list than when you use the FOR UPDATE CLAUSE OF clause with a
column list. Therefore, if you intend to update only a few columns of a table, your
program can run more efficiently if you include a column list.

The precompiler options NOFOR and STDSQL affect the use of the FOR UPDATE
clause in static SQL statements. For information on these options, see Table 48 on
page 400. If you do not specify the FOR UPDATE clause in a DECLARE CURSOR
statement, and you do not specify the STDSQL(YES) option or the NOFOR
precompiler options, you receive an error if you execute a positioned UPDATE
statement.

You can update a column of the identified table even though it is not part of the
result table. In this case, you do not need to name the column in the SELECT
statement. When the cursor retrieves a row (using FETCH) that contains a column
value you want to update, you can use UPDATE ... WHERE CURRENT OF to
identify the row that is to be updated.

> **_Read-only result table:_** Some result tables cannot be updated—for example, the result of joining two or more tables. Read-only result table specifications are described in greater detail in the discussion of DECLARE CURSOR in Chapter 5 of *DB2 SQL Reference*.

Step 2: Open the cursor

To tell DB2 that you are ready to process the first row of the result table, execute the OPEN statement in your program. DB2 then uses the SELECT statement within DECLARE CURSOR to identify a set of rows. If you use host variables in that SELECT statement, DB2 uses the *current value* of the variables to select the rows. The result table that satisfies the search condition might contain zero, one, or many rows. An example of an OPEN statement is:

```
EXEC SQL
  OPEN C1
END-EXEC.
```

If you use the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers in a cursor, DB2 determines the values in those special registers only when it opens the cursor. DB2 uses the values that it obtained at OPEN time for all subsequent FETCH statements.

Two factors that influence the amount of time that DB2 requires to process the OPEN statement are:
- Whether DB2 must perform any sorts before it can retrieve rows from the result table
- Whether DB2 uses parallelism to process the SELECT statement associated with the cursor

For more information, see "The effect of sorts on OPEN CURSOR" on page 706.

Step 3: Specify what to do at end-of-data

To determine if the program has retrieved the last row of data, test the SQLCODE field for a value of 100 or the SQLSTATE field for a value of '02000'. These codes occur when a FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH. For example:

```
IF SQLCODE = 100 GO TO DATA-NOT-FOUND.
```

An alternative to this technique is to code the WHENEVER NOT FOUND statement. The WHENEVER NOT FOUND statement can branch to another part of your program that issues a CLOSE statement. For example, to branch to label DATA-NOT-FOUND when the FETCH statement does not return a row, use this statement:

```
EXEC SQL
  WHENEVER NOT FOUND GO TO DATA-NOT-FOUND
END-EXEC.
```

Your program must anticipate and handle an end-of-data whenever you use a cursor to fetch a row. For further information about the WHENEVER NOT FOUND statement, see "Checking the execution of SQL statements" on page 74.

Step 4: Execute SQL statements

You execute one of these SQL statements using the cursor:
- A FETCH statement
- A positioned UPDATE statement

• A positioned DELETE statement

Using FETCH statements

Execute a FETCH statement for one of the following purposes:

• To copy data from a row of the result table into one or more host variables
• To position the cursor before you perform a positioned update or positioned delete operation

The following example shows a FETCH statement retrieves selected columns from the employee table:

```
EXEC SQL
  FETCH C1 INTO
    :HV-EMPNO, :HV-FIRSTNME, :HV-MIDINIT, :HV-LASTNAME, :HV-SALARY
  END-EXEC.
```

The SELECT statement within DECLARE CURSOR statement identifies the result table from which you fetch rows, but DB2 does not retrieve any data until your application program executes a FETCH statement.

When your program executes the FETCH statement, DB2 uses the cursor to point to a row in the result table. That row is called the *current row*. DB2 then copies the current row contents into the program host variables that you specified on the INTO clause of FETCH. This sequence repeats each time you issue FETCH, until you have processed all rows in the result table.

The row that DB2 points to when you execute a FETCH statement depends on whether the cursor is declared as a scrollable or non-scrollable. See "Scrollable and non-scrollable cursors" on page 85 for more information.

When you query a remote subsystem with FETCH, consider using block fetch for better performance. For more information see "Use block fetch" on page 383. Block fetch processes rows ahead of the current row. You cannot use a block fetch when you perform a positioned update or delete operation.

Using positioned UPDATE statements

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned UPDATE statement to modify the data in that row. An example of a positioned UPDATE statement is:

```
UPDATE DSN8710.EMP
    SET SALARY = 50000
    WHERE CURRENT OF C1
END-EXEC.
```

A positioned UPDATE statement updates the row that the cursor points to.

A positioned UPDATE statement must meet these conditions:

• You cannot update a row if your update violates any unique, check, or referential constraints.
• You cannot use an UPDATE statement to modify the rows of a created temporary table. However, you can use an UPDATE statement to modify the rows of a declared temporary table.
• If the right side of the SET clause in the UPDATE statement contains a fullselect, that fullselect cannot include a correlated name for a table that is being updated.

Using positioned DELETE statements

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned DELETE statement to delete that row. A example of a positioned DELETE statement looks like this:

```
EXEC SQL
  DELETE FROM DSN8710.EMP
    WHERE CURRENT OF C1
END-EXEC.
```

A positioned DELETE statement deletes the row that *cursor-name* points to.

A positioned DELETE statement must meet these conditions:

- You cannot use a DELETE statement with a cursor to delete rows from a created temporary table. However, you can use a DELETE statement with a cursor to delete rows from a declared temporary table.
- After you have deleted a row, you cannot update or delete another row using that cursor until you execute a FETCH statement to position the cursor on another row.
- You cannot delete a row if doing so violates any referential constraints.

Step 5: Close the cursor

If you finish processing the rows of the result table and you want to use the cursor again, issue a CLOSE statement to close the cursor. An example of a CLOSE statement looks like this:

```
EXEC SQL
  CLOSE C1
END-EXEC.
```

If you finish processing the rows of the result table, and you do not want to use the cursor, you can let DB2 automatically close the cursor when your program terminates.

Types of cursors

Cursors can be scrollable or not scrollable. They can also be held or not held. The following sections discuss these characteristics in more detail.

Scrollable and non-scrollable cursors

When you declare a cursor, you tell DB2 whether you want the cursor to be scrollable or non-scrollable by including or omitting the SCROLL clause. This clause determines whether the cursor moves sequentially forward through the result table or can move randomly through the result table.

Using a non-scrollable cursor

The simplest type of cursor is a non-scrollable cursor. A non-scrollable cursor always moves sequentially forward in the result table. When you open the cursor, the cursor is positioned before the first row in the result table. When you execute the first FETCH, the cursor is positioned on the first row. When you execute subsequent FETCH statements, the cursor moves one row ahead for each FETCH. After each FETCH statement, the cursor is positioned on the row that you fetched. After you execute a positioned UPDATE or positioned DELETE operation, the cursor stays at the current row of the result table. You cannot retrieve rows backward or move to a specific position in a result table with a non-scrollable cursor.

Using a scrollable cursor

To make a cursor scrollable, you declare it as scrollable. To use a scrollable cursor, you execute FETCH statements that indicate where you want to position the cursor.

If you want to order the rows of the cursor's result set, and you also want the cursor to be updatable, you need to declare the cursor as scrollable, even if you use it only to retrieve rows sequentially. You can use the ORDER BY clause in the declaration of an updatable cursor only if you declare the cursor as scrollable.

Declaring a scrollable cursor: To indicate that a cursor is scrollable, you declare it with the SCROLL keyword. The following examples show declarations for two basic forms of a scrollable cursor:

```
EXEC SQL DECLARE C1 INSENSITIVE SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8710.DEPT
  ORDER BY DEPTNO
  END-EXEC.
```

Figure 9. Declaration for an insensitive scrollable cursor

Figure 10. Declaration for a sensitive scrollable cursor

```
EXEC SQL DECLARE C2 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8710.DEPT
  ORDER BY DEPTNO
  END-EXEC.
```

Both of these examples show a characteristic of scrollable cursors: the *sensitivity*.

The first example shows a declaration for an *insensitive* scrollable cursor. Declaring a cursor with the INSENSITIVE keyword has the following effects:

* The size, the order of the rows, and the values for each row of the result table do not change after you open the cursor.
* The result table is read-only. Therefore, you cannot declare the cursor with the FOR UPDATE clause, and you cannot use the cursor for update or delete operations.

The second example shows a declaration for an *sensitive* scrollable cursor. Declaring a cursor as SENSITIVE has the following effects:

* When you execute positioned UPDATE and DELETE statements with the cursor, those updates are visible in the result table.
* When the current value of a row no longer satisfies the SELECT statement for the cursor, that row is no longer visible in the result table.
* When a row of the result table is deleted from the underlying table, the row is no longer visible in the result table.
* Changes that are made to the underlying table by other cursors or other application processes can be visible in the result table, depending on whether the FETCH statements that you use with the cursor are FETCH INSENSITIVE or FETCH SENSITIVE statements.

In DB2 Version 7, when you declare a cursor as SENSITIVE, you must also declare it as STATIC. Declaring the cursor as STATIC has the following effects:

* The size of the result table does not grow after you open the cursor.

Rows that are inserted into the underlying table are not added to the result table.

- The order of the rows does not change after you open the cursor.

    If the cursor declaration contains an ORDER BY clause, and columns that are in the ORDER BY clause are updated after you open the cursor, the order of rows in the result table does not change.

Determining attributes of a cursor by checking the SQLCA: After you open a cursor, you can determine the following attributes of the cursor by checking the SQLWARN1, SQLWARN4, and SQLWARN5 fields of the SQLCA:

SQLWARN1
  Whether the cursor is scrollable or non-scrollable

SQLWARN4
  Whether the cursor is sensitive or insensitive

SQLWARN5
  Whether the cursor is read-only, readable and deletable, or readable, deletable, and updatable

If the OPEN statement executes with no errors or warnings, DB2 does not set SQLWARN0 when it sets SQLWARN1, SQLWARN4, or SQLWARN5. See Appendix C of *DB2 SQL Reference* for specific information on fields in the SQLCA.

Retrieving rows with a scrollable cursor: When you open any cursor, the cursor is positioned before the first row of the result table. You move a scrollable cursor around in the result table by specifying a *fetch orientation* keyword in a FETCH statement. A fetch orientation keyword indicates the absolute or relative position of the cursor when the FETCH statement is executed. Table 4 lists the fetch orientation keywords that you can specify and their meanings.

Table 4. Positions for a scrollable cursor

Keyword in FETCH statement	Cursor position when the FETCH is executed
BEFORE	Before the first row
FIRST	At the first row
LAST or ABSOLUTE −1	At the last row
AFTER	After the last row
ABSOLUTE[1]	To an absolute row number. +1 is the first row.
RELATIVE	Forward or backward a relative number of rows
CURRENT	At the current row
PRIOR or RELATIVE −1	To the previous row
NEXT	To the next row (default)

Note:

1. FETCH ABSOLUTE 0 positions the cursor before the first row, as does FETCH BEFORE. However, FETCH ABSOLUTE 0 returns SQLCODE +100, but FETCH BEFORE returns SQLCODE 0.

For example, to use the cursor that is declared in Figure 9 on page 86 to fetch the fifth row of the result table, use a FETCH statement like this:

```
EXEC SQL FETCH ABSOLUTE +5 C1 INTO :HVDEPTNO, :DEPTNAME, :MGRNO;
```

When you declare a cursor as SENSITIVE, changes that other processes or cursors make to the underlying table *can be* be visible to the result table of your

cursor. Whether those changes *are* visible depends on whether you specify the SENSITIVE or INSENSITIVE keyword when you execute FETCH statements with the cursor. When you specify FETCH INSENSITIVE, changes that other processes or cursors make to the underlying table are not visible in the result table. When you specify FETCH SENSITIVE, changes that other processes or cursors make to the underlying table are visible in the result table. Table 5 summarizes the sensitivity values and their effects on the result table of a scrollable cursor.

Table 5. How sensitivity affects the result table for a scrollable cursor

DECLARE sensitivity	**FETCH sensitivity**	
	INSENSITIVE	**SENSITIVE**
INSENSITIVE	No changes to the underlying table are visible in the result table. Positioned UPDATE and DELETE statements using the cursor are not allowed.	Not valid.
SENSITIVE	Only changes that are made by the cursor are visible in the result table.	All changes are visible in the result table.

Determining the number of rows in the result table for a scrollable cursor:
You can determine how many rows are in the result table of an INSENSITIVE or SENSITIVE STATIC scrollable cursor. To do that, execute a FETCH statement, such as FETCH AFTER, that positions the cursor after the last row. Then examine the SQLCA. Fields SQLERRD(1) and SQLERRD(2) (sqlerrd[0] and sqlerrd[1] for C and C⁺⁺) contain the number of rows in the result table.

Holes in the result table: Scrollable cursors that are declared as INSENSITIVE or SENSITIVE STATIC follow a *static model*, which means that DB2 determines the size of the result table and the order of the rows when you open the cursor. Updating or deleting rows from the underlying table after the cursor is open can result in *holes* in the result table. A hole in the result table occurs when a delete or update operation results in a difference between the result table and the underlying base table.

The following examples demonstrate how holes can occur.

Example: Creating a delete hole: Suppose that table A consists of one integer column, COL1, which has the following values:
Now suppose that you declare the following cursor, which you use to delete rows

```
1
2
3
4
5
```

from A:
```
EXEC SQL DECLARE C3 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT COL1
  FROM A
  FOR UPDATE OF COL1;
```

Now you execute the following SQL statements:

```
EXEC SQL OPEN C3;
EXEC SQL FETCH ABSOLUTE +3 C3 INTO :HVCOL1;
EXEC SQL DELETE FROM A WHERE CURRENT OF C3;
```

The positioned delete statement creates a *delete hole*, as shown in Figure 11.
After you execute the positioned delete statement, the third row is deleted from the
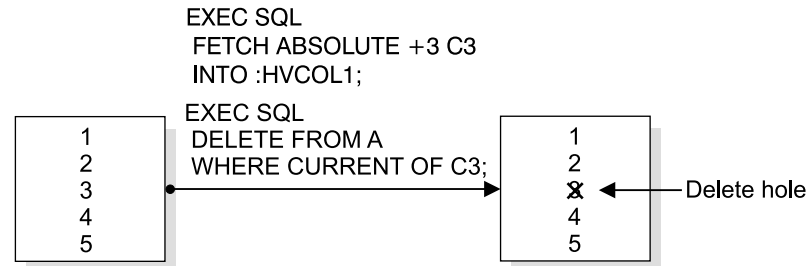
Figure 11. Creating a delete hole

result table, but the result table does not shrink to fill the space that the deleted row
creates.

Example: Creating an update hole: Suppose that you declare the following cursor,
which you use to update rows in A:
```
EXEC SQL DECLARE C4 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT COL1
  FROM A
  WHERE COL1<6;
```

Now you execute the following SQL statements:
```
EXEC SQL OPEN C4;
UPDATE A SET COL1=COL1+1;
```

The searched UPDATE statement creates an *update hole*, as shown in Figure 12.
After you execute the searched UPDATE statement, the last row no longer qualifies
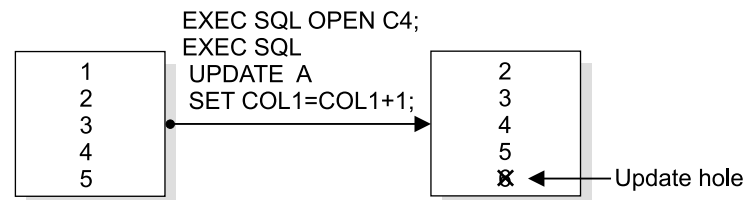
Figure 12. Creating an update hole

for the result table, but the result table does not shrink to fill the space that the
disqualified row creates.

If you try to fetch from an delete hole, DB2 issues an SQL warning. If you try to
update or delete the delete hole, DB2 issues an SQL error. You can remove a
delete hole only by opening the scrollable cursor, setting a savepoint, executing a
positioned DELETE statement with the scrollable cursor, and rolling back to the
savepoint.

If you try to fetch from an update hole, DB2 issues an SQL warning. If you try to
delete the update hole, DB2 issues an SQL error. However, you can convert an

update hole back to a result table row by updating the row in the base table, as shown in Figure 13. You can update the base table with a searched UPDATE statement in the same application process, or a searched or positioned UPDATE statement in another application process. After you update the base table , if the row qualifies for the result table, the update hole disappears.
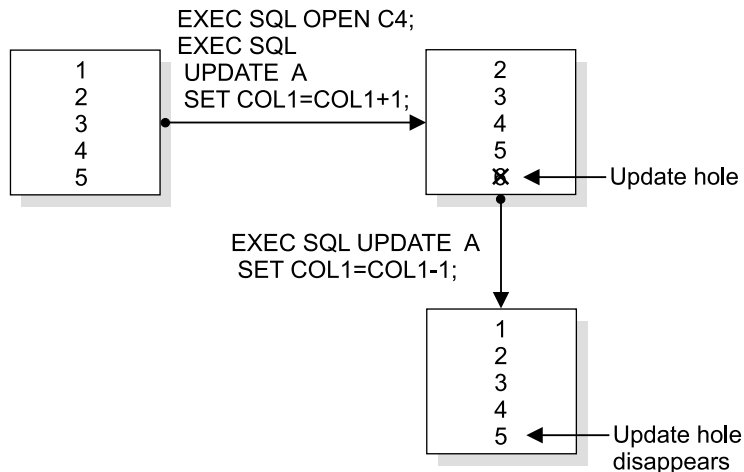
Figure 13. Removing an update hole

A hole becomes visible to a cursor when a cursor operation returns a non-zero SQLCODE. The point at which a hole becomes visible depends on the following factors:

- Whether the scrollable cursor creates the hole
- Whether the FETCH statement is FETCH SENSITIVE or FETCH INSENSITIVE

If the scrollable cursor creates the hole, the hole is visible when you execute a FETCH statement for the row that contains the hole. The FETCH statement can be FETCH INSENSITIVE or FETCH SENSITIVE.

If an update or delete operation outside the scrollable cursor creates the hole, the hole is visible at the following times:

- If you execute a FETCH SENSITIVE statement for the row that contains the hole, the hole is visible when you execute the FETCH statement.
- If you execute a FETCH INSENSITIVE statement, the hole is not visible when you execute the FETCH statement. DB2 returns the row as it was before the update or delete operation occurred. However, if you follow the FETCH INSENSITIVE statement with a positioned UPDATE or DELETE statement, the hole becomes visible.

Creating declared temporary tables for scrollable cursors

DB2 uses declared temporary tables for processing scrollable cursors. Therefore, before you can use a scrollable cursor, your database administrator needs to create a TEMP database and TEMP table spaces for those declared temporary tables. If there is more than one TEMP table space in the subsystem, DB2 chooses the table spaces to use for scrollable cursors.

The page size of the TEMP table space must be large enough to hold the longest row in the declared temporary table. See Part 2 of *DB2 Installation Guide* for information on calculating the page size for TEMP table spaces that are used for scrollable cursors.

Held and non-held cursors

When you declare a cursor, you tell DB2 whether you want the cursor to be held or not held by including or omitting the WITH HOLD clause. A held cursor, which is declare WITH HOLD, does not close after a commit operation. A cursor that is not held closes after a commit operation.

After a commit operation, a held cursor is positioned after the last row retrieved and before the next logical row of the result table to be returned.

A held cursor can close when:
- You issue a CLOSE cursor, ROLLBACK, or CONNECT statement
- You issue a CAF CLOSE function call or an RRSAF TERMINATE THREAD function call
- The application program terminates.

If the program abnormally terminates, the cursor position is lost. To prepare for restart, your program must reposition the cursor.

The following restrictions apply to cursors that are declared WITH HOLD:
    Do not use DECLARE CURSOR WITH HOLD with the new user signon from a DB2 attachment facility, because all open cursors are closed.
    Do not declare a WITH HOLD cursor in a thread that could become inactive. If you do, its locks are held indefinitely.

IMS

You *cannot* use DECLARE CURSOR...WITH HOLD in message processing programs (MPP) and message-driven batch message processing (BMP). Each message is a new user for DB2; whether or not you declare them using WITH HOLD, no cursors continue for new users. You can use WITH HOLD in non-message-driven BMP and DL/I batch programs.

CICS

In CICS applications, you can use DECLARE CURSOR...WITH HOLD to indicate that a cursor should not close at a commit or sync point. However, SYNCPOINT ROLLBACK closes all cursors, and end-of-task (EOT) closes all cursors before DB2 reuses or terminates the thread. Because pseudo-conversational transactions usually have multiple EXEC CICS RETURN statements and thus span multiple EOTs, the scope of a held cursor is limited. Across EOTs, you must reopen and reposition a cursor declared WITH HOLD, as if you had not specified WITH HOLD.

You should always close cursors that you no longer need. If you let DB2 close a CICS attachment cursor, the cursor might not close until the CICS attachment facility reuses or terminates the thread.

The following cursor declaration causes the cursor to maintain its position in the DSN8710.EMP table after a commit point:

```
EXEC SQL
  DECLARE EMPLUPDT CURSOR WITH HOLD FOR
    SELECT EMPNO, LASTNAME, PHONENO, JOB, SALARY, WORKDEPT
```

```
              FROM DSN8710.EMP
              WHERE WORKDEPT < 'D11'
              ORDER BY EMPNO
          END-EXEC.
```

Examples of using cursors

The following examples show the SQL statements that you must include in a
COBOL program to define and use a cursor.

Figure 14 on page 93 shows how to use a non-scrollable cursor to perform FETCH
and positioned UPDATE operations.

```
***************************************************
* Declare a cursor that will be used to update    *
* the JOB and WORKDEPT columns of the EMP table.  *
***************************************************
 EXEC SQL
  DECLARE THISEMP CURSOR FOR
     SELECT EMPNO, LASTNAME,
        WORKDEPT, JOB
     FROM DSN8710.EMP
     WHERE WORKDEPT = 'D11'
   FOR UPDATE OF JOB
 END-EXEC.
***************************************************
* Open the cursor                                 *
***************************************************
 EXEC SQL
  OPEN THISEMP
 END-EXEC.
***************************************************
* Indicate what action to take when all rows      *
* in the result table have been fetched.          *
***************************************************
 EXEC SQL
  WHENEVER NOT FOUND
     GO TO CLOSE-THISEMP
 END-EXEC.
***************************************************
* Fetch a row to position the cursor.             *
***************************************************
 EXEC SQL
  FETCH THISEMP
     INTO :EMP-NUM, :NAME2,
        :DEPT, :JOB-NAME
 END-EXEC.
***************************************************
* Update the row that the cursor points to.       *
***************************************************
 EXEC SQL
  UPDATE DSN8710.EMP
     SET JOB = :NEW-JOB
     WHERE CURRENT OF THISEMP
 END-EXEC.
***************************************************
* Branch back to fetch and process the next row.  *
***************************************************
 :
***************************************************
* Close the cursor                                *
***************************************************
 CLOSE-THISEMP.
  EXEC SQL
     CLOSE THISEMP
  END-EXEC.
```

Figure 14. Performing cursor operations with a non-scrollable cursor

Figure 15 on page 94 shows how to use a scrollable cursor to retrieve data backward from a table.

```
***************************************************
* Declare a cursor that will be used to retrieve *
* the data backward from the EMP table. The      *
* cursor should have access to changes by other  *
* processes.                                      *
***************************************************
 EXEC SQL
  DECLARE THISEMP SENSITIVE STATIC SCROLL CURSOR FOR
    SELECT EMPNO, LASTNAME, WORKDEPT, JOB
     FROM DSN8710.EMP
 END-EXEC.
***************************************************
* Open the cursor                                 *
***************************************************
 EXEC SQL
  OPEN THISEMP
 END-EXEC.
***************************************************
* Indicate what action to take when all rows      *
* in the result table have been fetched.          *
***************************************************
 EXEC SQL
  WHENEVER NOT FOUND GO TO CLOSE-THISEMP
 END-EXEC.
***************************************************
* Position the cursor after the last row of the   *
* result table. This FETCH statement cannot       *
* include the SENSITIVE or INSENSITIVE keyword    *
* and cannot contain an INTO clause.              *
***************************************************
 EXEC SQL
  FETCH AFTER THISEMP
 END-EXEC.
***************************************************
* Fetch the previous row in the table.            *
***************************************************
 EXEC SQL
  FETCH SENSITIVE PRIOR THISEMP
    INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
 END-EXEC.
***************************************************
* Check that the fetched row is not a hole        *
* (SQLCODE +222). If the row is not a             *
* hole, print the row contents.                   *
***************************************************
 IF SQLCODE IS GREATER THAN OR EQUAL TO 0 AND
    SQLCODE IS NOT EQUAL TO +100 AND
    SQLCODE IS NOT EQUAL TO +222 AND THEN
   PERFORM PRINT-RESULTS.
***************************************************
* Branch back to fetch the previous row.          *
***************************************************
 :
 :
***************************************************
* Close the cursor                                *
***************************************************
 CLOSE-THISEMP.
  EXEC SQL
    CLOSE THISEMP
  END-EXEC.
```

Figure 15. Performing cursor operations with a scrollable cursor

Chapter 8. Generating declarations for your tables using DCLGEN

DCLGEN, the declarations generator supplied with DB2, produces a DECLARE statement you can use in a C, COBOL, or PL/I program, so that you do not need to code the statement yourself. For detailed syntax of DCLGEN, see Chapter 2 of *DB2 Command Reference*.

DCLGEN generates a table declaration and puts it into a member of a partitioned data set that you can include in your program. When you use DCLGEN to generate a table's declaration, DB2 gets the relevant information from the DB2 catalog, which contains information about the table's definition and the definition of each column within the table. DCLGEN uses this information to produce a complete SQL DECLARE statement for the table or view and a matching PL/I, or C structure declaration or COBOL record description. You can use DCLGEN for table declarations only if the table you are declaring already exists.

You must use DCLGEN before you precompile your program. Supply DCLGEN with the table or view name before you precompile your program. To use the declarations generated by DCLGEN in your program, use the SQL INCLUDE statement.

DB2 must be active before you can use DCLGEN. You can start DCLGEN in several different ways:

* From ISPF through DB2I. Select the DCLGEN option on the DB2I Primary Option Menu panel. Next, fill in the DCLGEN panel with the information it needs to build the declarations. Then press ENTER.
* Directly from TSO. To do this, sign on to TSO, issue the TSO command DSN, and then issue the subcommand DCLGEN.
* From a CLIST, running in TSO foreground or background, that issues DSN and then DCLGEN.
* With JCL. Supply the required information, using JCL, and run DCLGEN in batch.

  If you wish to start DCLGEN in the foreground, and your table names include DBCS characters, you must input and display double-byte characters. If you do not have a terminal that displays DBCS characters, you can enter DBCS characters using the hex mode of ISPF edit.

Invoking DCLGEN through DB2I

The easiest way to start DCLGEN is through DB2I. Figure 16 on page 96 shows the DCLGEN panel you reach by selecting option 2, DCLGEN, on the DB2I Primary Option Menu. For more instructions on using DB2I, see "Using ISPF and DB2 Interactive (DB2I)" on page 431.

```
 DSNEDP01                    DCLGEN                        SSID: DSN
 ===>

 Enter table name for which declarations are required:
  1   SOURCE TABLE NAME ===>                      (Unqualified table name)
  2   TABLE OWNER     ===>                                  (Optional)
  3   AT LOCATION ..... ===>                               (Optional)

 Enter destination data set:        (Can be sequential or partitioned)
  4   DATA SET NAME ... ===>
  5   DATA SET PASSWORD ===>         (If password protected)

 Enter options as desired:
  6   ACTION .......... ===>         (ADD new or REPLACE old declaration)
  7   COLUMN LABEL .... ===>         (Enter YES for column label)
  8   STRUCTURE NAME .. ===>                                (Optional)
  9   FIELD NAME PREFIX ===>                                (Optional)
 10   DELIMIT DBCS .... ===>         (Enter YES to delimit DBCS identifiers)
 11   COLUMN SUFFIX ... ===>         (Enter YES to append column name)
 12   INDICATOR VARS .. ===>         (Enter YES for indicator variables)


 PRESS: ENTER to process    END to exit      HELP for more information
```

Figure 16. DCLGEN panel

Fill in the DCLGEN panel as follows:

1 SOURCE TABLE NAME

Is the unqualified name of the table, view, or created temporary table for which you want DCLGEN to produce SQL data declarations. The table can be stored at your DB2 location or at another DB2 location. To specify a table name at another DB2 location, enter the table qualifier in the TABLE OWNER field and the location name in the AT LOCATION field. DCLGEN generates a three-part table name from the SOURCE TABLE NAME, TABLE OWNER, and AT LOCATION fields. You can also use an alias for a table name.

To specify a table name that contains special characters or blanks, enclose the name in apostrophes. If the name contains apostrophes, you must double each one("). For example, to specify a table named DON'S TABLE, enter the following:

```
'DON''S TABLE'
```

You do not have to enclose DBCS table names in apostrophes. If you do not enclose the table name in apostrophes, DB2 translates lowercase characters to uppercase.

DCLGEN does not treat the underscore as a special character. For example, the table name JUNE_PROFITS does not need to be enclosed in apostrophes. Because COBOL field names cannot contain underscores, DCLGEN substitutes hyphens (-) for single-byte underscores in COBOL field names built from the table name.

2 TABLE OWNER

Is the owner of the source table. If you do not specify this value and the table is a local table, DB2 assumes that the table qualifier is your TSO logon ID. If the table is at a remote location, you must specify this value.

3 AT LOCATION

Is the location of a table or view at another DB2 subsystem. If you specify this parameter, you must also specify a qualified name in the SOURCE TABLE NAME field. The value of the AT LOCATION field prefixes the table name on the SQL DECLARE statement as follows:

`location_name.owner_id.table_name`

For example, for the location PLAINS_GA:

`PLAINS_GA.CARTER.CROP_YIELD_89`

If you do not specify a location, then this option defaults to the local location name. This field applies to DB2 private protocol access only (that is, the location you name must be another DB2 for OS/390 and z/OS).

4 DATA SET NAME

Is the name of the data set you allocated to contain the declarations that DCLGEN produces. You must supply a name; there is no default.

The data set must already exist, be accessible to DCLGEN, and can be either sequential or partitioned. If you do not enclose the data set name in apostrophes, DCLGEN adds a standard TSO prefix (user ID) and suffix (language). DCLGEN knows what the host language is from the DB2I defaults panel.

For example, for library name LIBNAME(MEMBNAME), the name becomes:

`userid.libname.language(membname)`

and for library name LIBNAME, the name becomes:

`userid.libname.language`

If this data set is password protected, you must supply the password in the DATA SET PASSWORD field.

5 DATA SET PASSWORD

Is the password for the data set in the DATA SET NAME field, if the data set is password protected. It does not display on your terminal, and is not recognized if you issued it from a previous session.

6 ACTION

Tells DCLGEN what to do with the output when it is sent to a partitioned data set. (The option is ignored if the data set you specify in DATA SET NAME field is sequential.)

ADD indicates that an old version of the output does not exist, and creates a new member with the specified data set name. This is the default.

REPLACE replaces an old version, if it already exists. If the member does not exist, this option creates a new member.

7 COLUMN LABEL

Tells DCLGEN whether to include labels declared on any columns of the table or view as comments in the data declarations. (The SQL statement LABEL ON creates column labels to use as supplements to column names.) Use:

YES to include column labels.

NO to ignore column labels. This is the default.

8 STRUCTURE NAME

Is the name of the generated data structure. The name can be up to 31 characters. If the name is not a DBCS string, and the first character is not

alphabetic, then enclose the name in apostrophes. If you use special characters, be careful to avoid name conflicts.

If you leave this field blank, DCLGEN generates a name that contains the table or view name with a prefix of *DCL*. If the language is COBOL or PL/I, and the table or view name consists of a DBCS string, the prefix consists of DBCS characters.

C language characters you enter in this field do not fold to uppercase.

9 FIELD NAME PREFIX

Specifies a prefix that DCLGEN uses to form field names in the output. For example, if you choose ABCDE, the field names generated are ABCDE1, ABCDE2, and so on.

DCLGEN accepts a field name prefix of up to 28 bytes that can include special and double-byte characters. If you specify a single-byte or mixed-string prefix and the first character is not alphabetic, apostrophes must enclose the prefix. If you use special characters, be careful to avoid name conflicts.

For COBOL and PL/I, if the name is a DBCS string, DCLGEN generates DBCS equivalents of the suffix numbers. For C, characters you enter in this field do not fold to uppercase.

If you leave this field blank, the field names are the same as the column names in the table or view.

10 DELIMIT DBCS

Tells DCLGEN whether to delimit DBCS table names and column names in the table declaration. Use:
YES to enclose the DBCS table and column names with SQL delimiters.
NO to not delimit the DBCS table and column names.

11 COLUMN SUFFIX

Tells DCLGEN whether to form field names by attaching the column name as a suffix to value you specify in FIELD NAME PREFIX. For example, if you specify YES, the field name prefix is NEW, and the column name is EMPNO, then the field name is NEWEMPNO.

If you specify YES, you must also enter a value in FIELD NAME PREFIX. If you do not enter a field name prefix, DCLGEN issues a warning message and uses the column names as the field names.

The default is NO, which does not use the column name as a suffix, and allows the value in FIELD NAME PREFIX to control the field names, if specified.

12 INDICATOR VARS

Tells DCLGEN whether to generate an array of indicator variables for the host variable structure.

If you specify YES, the array name is the table name with a prefix of "I" (or DBCS letter "<I>" if the table name consists solely of double-byte characters). The form of the data declaration depends on the language:
For a C program: `short int I`*table-name*`[`*n*`];`
For a COBOL program: `01 I`*table-name* `PIC S9(4) USAGE COMP OCCURS` *n* `TIMES.`
For a PL/I program: `DCL I`*table-name*`(`*n*`) BIN FIXED(15);`

where *n* is the number of columns in the table. For example, if you define a table:

```
CREATE TABLE HASNULLS (CHARCOL1 CHAR(1), CHARCOL2 CHAR(1));
```

and you request an array of indicator variables for a COBOL program, DCLGEN might generate the following host variable declaration:

```
01  DCLHASNULLS.
    10 CHARCOL1              PIC X(1).
    10 CHARCOL2              PIC X(1).
01  IHASNULLS PIC S9(4) USAGE COMP OCCURS 2 TIMES.
```

The default is NO, which does not generate an indicator variable array.

DCLGEN generates a table or column name in the DECLARE statement as a non-delimited identifier unless at least one of the following is true:
• The name contains special characters and is not a DBCS string.
• The name is a DBCS string, and you have requested delimited DBCS names.

If you are using an SQL reserved word as an identifier, you must edit the DCLGEN output in order to add the appropriate SQL delimiters.

Including the data declarations in your program

Use the following SQL INCLUDE statement to place the generated table declaration and COBOL record description in your source program:

```
EXEC SQL
   INCLUDE member-name
END-EXEC.
```

For example, to include a description for the table DSN8710.EMP, code:

```
EXEC SQL
   INCLUDE DECEMP
END-EXEC.
```

In this example, DECEMP is a name of a member of a partitioned data set that contains the table declaration and a corresponding COBOL record description of the table DSN8710.EMP. (A COBOL record description is a two-level host structure that corresponds to the columns of a table's row. For information on host structures, see "Chapter 9. Embedding SQL statements in host languages" on page 107.) To get a current description of the table, use DCLGEN to generate the table's declaration and store it as member DECEMP in a library (usually a partitioned data set) just before you precompile the program.

DCLGEN produces output that is intended to meet the needs of most users, but occasionally, you will need to edit the DCLGEN output to work in your specific case. For example, DCLGEN is unable to determine whether a column defined as NOT NULL also contains the DEFAULT clause, so you must edit the DCLGEN output to add the DEFAULT clause to the appropriate column definitions.

DCLGEN support of C, COBOL, and PL/I languages

DCLGEN derives variable names from the source in the database. In Table 6 on page 100, *var* represents variable names that DCLGEN provides when it is necessary to clarify the host language declaration.

Table 6. Declarations generated by DCLGEN

SQL Data Type[6]	C	COBOL	PL/I
SMALLINT	short int	PIC S9(4) USAGE COMP	BIN FIXED(15)
INTEGER	long int	PIC S9(9) USAGE COMP	BIN FIXED(31)
DECIMAL(p,s) or NUMERIC(p,s)	decimal(p,s)[4]	PIC S9(p-s)V9(s) USAGE COMP-3	DEC FIXED(p,s) If p>15, a warning is generated.
REAL or FLOAT(n) 1 <= n <= 21	float	USAGE COMP-1	BIN FLOAT(n)
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	double	USAGE COMP-2	BIN FLOAT(n)
CHAR(1)	char	PIC X(1)	CHAR(1)
CHAR(n)	char var [n+1]	PIC X(n)	CHAR(n)
VARCHAR(n)	struct {short int var_len; char var_data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC X(n).	CHAR(n) VAR
GRAPHIC(1)	wchar_t	PIC G(1)	GRAPHIC(1)
GRAPHIC(n) n > 1	wchar_t var[n+1];	PIC G(n) USAGE DISPLAY-1.[1] or PIC N(n).[1]	GRAPHIC(n)
VARGRAPHIC(n)	struct VARGRAPH {short len; wchar_t data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC G(n) USAGE DISPLAY-1.[1] or 10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC N(n).[1]	GRAPHIC(n) VAR
BLOB(n)[5]	SQL TYPE IS BLOB_LOCATOR	USAGE SQL TYPE IS BLOB-LOCATOR	SQL TYPE IS BLOB_LOCATOR
CLOB(n)[5]	SQL TYPE IS CLOB_LOCATOR	USAGE SQL TYPE IS CLOB-LOCATOR	SQL TYPE IS CLOB_LOCATOR
DBCLOB(n)[5]	SQL TYPE IS DBCLOB_LOCATOR	USAGE SQL TYPE IS DBCLOB-LOCATOR	SQL TYPE IS DBCLOB_LOCATOR
ROWID	SQL TYPE IS ROWID	USAGE SQL TYPE IS ROWID	SQL TYPE IS ROWID
DATE	char var[11][2]	PIC X(10)[2]	CHAR(10)[2]
TIME	char var[9][3]	PIC X(8)[3]	CHAR(8)[3]
TIMESTAMP	char var[27]	PIC X(26)	CHAR(26)

Notes to Table 6:

1. DCLGEN chooses the format based on the character you specify as the DBCS symbol on the COBOL Defaults panel.
2. This declaration is used unless there is a date installation exit for formatting dates, in which case the length is that specified for the LOCAL DATE LENGTH installation option.
3. This declaration is used unless there is a time installation exit for formatting times, in which case the length is that specified for the LOCAL TIME LENGTH installation option.
4. If your C compiler does not support the decimal data type, edit your DCLGEN output and replace the decimal data declarations with declarations of type double.
5. For a BLOB, CLOB, or DBCLOB data type, DCLGEN generates a LOB locator.
6. For a distinct type, DCLGEN generates the host language equivalent of the source data type.

For further details about the DCLGEN subcommand, see Chapter 2 of *DB2 Command Reference*.

Example: Adding a table declaration and host-variable structure to a library

This example adds an SQL table declaration and a corresponding host-variable structure to a library. This example is based on the following scenario:
- The library name is *prefix*.TEMP.COBOL.
- The member is a new member named VPHONE.
- The table is a local table named DSN8710.VPHONE.
- The host-variable structure is for COBOL.
- The structure receives the default name DCLVPHONE.

Information you must enter is in bold-faced type.

Step 1. Specify COBOL as the host language

Select option **D** on the ISPF/PDF menu to display the DB2I Defaults panel.

Specify **COBOL** as the application language as shown in Figure 17 on page 102, and then press Enter. The COBOL Defaults panel then displays as shown in Figure 18 on page 102.

Fill in the COBOL defaults panel as necessary. Press Enter to save the new defaults, if any, and return to the DB2I Primary Option menu.

```
DSNEOP01                    DB2I DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  DB2 NAME ............. ===> DSN        (Subsystem identifier)
 2  DB2 CONNECTION RETRIES ===> 0          (How many retries for DB2 connection)
 3  APPLICATION LANGUAGE   ===> COBOL      (ASM, C, CPP, COBOL, COB2, IBMCOB,
                                            FORTRAN,PLI)
 4  LINES/PAGE OF LISTING  ===> 80         (A number from 5 to 999)
 5  MESSAGE LEVEL ........ ===> I          (Information, Warning, Error, Severe)
 6  SQL STRING DELIMITER   ===> DEFAULT    (DEFAULT, ' or ")
 7  DECIMAL POINT ........ ===> .          (. or ,)
 8  STOP IF RETURN CODE >= ===> 8          (Lowest terminating return code)
 9  NUMBER OF ROWS         ===> 20         (For ISPF Tables)
10  CHANGE HELP BOOK NAMES?===> NO         (YES to change HELP data set names)
11  DB2I JOB STATEMENT:   (Optional if your site has a SUBMIT exit)
    ===> //USRT001A JOB (ACCOUNT),'NAME'
    ===> //*
    ===> //*
    ===> //*

PRESS: ENTER to process      END to cancel    HELP for more information
```

Figure 17. DB2I defaults panel—changing the application language

```
DSNEOP02                    COBOL DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  COBOL STRING DELIMITER ===>            (DEFAULT, ' or ")
 2  DBCS SYMBOL FOR DCLGEN ===>            (G/N - Character in PIC clause)
```

Figure 18. The COBOL defaults panel. Shown only if the field APPLICATION LANGUAGE on the DB2I Defaults panel is COBOL, COB2, or IBMCOB.

Step 2. Create the table declaration and host structure

Select option **2** on the DB2I Primary Option menu, and press Enter to display the DCLGEN panel.

Fill in the fields as shown in Figure 19 on page 103, and then press Enter.

```
DSNEDP01                    DCLGEN                         SSID: DSN
===>
Enter table name for which declarations are required:

 1   SOURCE TABLE NAME ===> DSN8710.VPHONE
 2   TABLE OWNER      ===>
 3   AT LOCATION ..... ===>            (Location of table, optional)

Enter destination data set:           (Can be sequential or partitioned)
 4   DATA SET NAME ... ===> TEMP(VPHONEC)
 5   DATA SET PASSWORD ===>            (If password protected)

Enter options as desired:
 6   ACTION .......... ===> ADD        (ADD new or REPLACE old declaration)
 7   COLUMN LABEL .... ===> NO         (Enter YES for column label)
 8   STRUCTURE NAME .. ===>                        (Optional)
 9   FIELD NAME PREFIX ===>                        (Optional)
10   DELIMIT DBCS     ===> YES         (Enter YES to delimit DBCS identifiers)
11   COLUMN SUFFIX ... ===> NO         (Enter YES to append column name)
12   INDICATOR VARS .. ===> NO         (Enter YES for indicator variables)


PRESS: ENTER to process    END to exit     HELP for more information
```

Figure 19. DCLGEN panel—selecting source table and destination data set

If the operation succeeds, a message displays at the top of your screen as shown in Figure 20.

```
DSNE905I EXECUTION COMPLETE, MEMBER VPHONEC ADDED
***
```

Figure 20. Successful completion message

DB2 then displays the screen as shown in Figure 21 on page 104. Press Enter to return to the DB2I Primary Option menu.

```
DSNEDP01                     DCLGEN                          SSID: DSN
===>
DSNE294I SYSTEM RETCODE=000      USER OR DSN RETCODE=0
Enter table name for which declarations are required:
 1  SOURCE TABLE NAME ===> DSN8710.VPHONE
 2  TABLE OWNER      ===>
 3  AT LOCATION ..... ===>              (Location of table, optional)

Enter destination data set:           (Can be sequential or partitioned)
 4  DATA SET NAME ... ===> TEMP(VPHONEC)
 5  DATA SET PASSWORD ===>              (If password protected)


Enter options as desired:
 6  ACTION .......... ===> ADD          (ADD new or REPLACE old declaration)
 7  COLUMN LABEL .... ===> NO           (Enter YES for column label)
 8  STRUCTURE NAME .. ===>                          (Optional)
 9  FIELD NAME PREFIX ===>                          (Optional)
10  DELIMIT DBCS     ===>               (Enter YES to delimit DBCS identifiers)
11  COLUMN SUFFIX ... ===>              (Enter YES to append column name)
12  INDICATOR VARS .. ===>              (Enter YES for indicator variables)



PRESS: ENTER to process    END to exit      HELP for more information
```

Figure 21. DCLGEN panel—displaying system and user return codes

Step 3. Examine the results

To browse or edit the results, first exit from DB2I by entering **X** on the command line of the DB2I Primary Option menu. The ISPF/PDF menu is then displayed, and you can select either the browse or the edit option to view the results.

For this example, the data set to edit is *prefix*.TEMP.COBOL(VPHONEC), which is shown in Figure 22 on page 105.

```
***** DCLGEN TABLE(DSN8710.VPHONE)                            ***
*****       LIBRARY(SYSADM.TEMP.COBOL(VPHONEC))               ***
*****       QUOTE                                             ***
***** ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS ***
        EXEC SQL DECLARE DSN8710.VPHONE TABLE
        ( LASTNAME                       VARCHAR(15) NOT NULL,
          FIRSTNAME                      VARCHAR(12) NOT NULL,
          MIDDLEINITIAL                  CHAR(1) NOT NULL,
          PHONENUMBER                    VARCHAR(4) NOT NULL,
          EMPLOYEENUMBER                 CHAR(6) NOT NULL,
          DEPTNUMBER                     CHAR(3) NOT NULL,
          DEPTNAME                       VARCHAR(36) NOT NULL
        ) END-EXEC.
***** COBOL DECLARATION FOR TABLE DSN8710.VPHONE              ******
     01  DCLVPHONE.
         10 LASTNAME.
            49 LASTNAME-LEN      PIC S9(4) USAGE COMP.
            49 LASTNAME-TEXT     PIC X(15).
         10 FIRSTNAME.
            49 FIRSTNAME-LEN     PIC S9(4) USAGE COMP.
            49 FIRSTNAME-TEXT    PIC X(12).
         10 MIDDLEINITIAL        PIC X(1).
         10 PHONENUMBER.
            49 PHONENUMBER-LEN   PIC S9(4) USAGE COMP.
            49 PHONENUMBER-TEXT  PIC X(4).
         10 EMPLOYEENUMBER       PIC X(6).
         10 DEPTNUMBER           PIC X(3).
         10 DEPTNAME.
            49 DEPTNAME-LEN      PIC S9(4) USAGE COMP.
            49 DEPTNAME-TEXT     PIC X(36).
***** THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7  ******
```

Figure 22. DCLGEN results displayed in edit mode

Chapter 9. Embedding SQL statements in host languages

This chapter provides detailed information about using each of the following languages to write embedded SQL application programs:
- "Coding SQL statements in an assembler application"
- "Coding SQL statements in a C or a C⁺⁺ application" on page 121
- "Coding SQL statements in a COBOL application" on page 141
- "Coding SQL statements in a FORTRAN application" on page 164
- "Coding SQL statements in a PL/I application" on page 174.
- "Coding SQL statements in a REXX application" on page 189.

For each language, there are unique instructions or details about:
- Defining the SQL communications area
- Defining SQL descriptor areas
- Embedding SQL statements
- Using host variables
- Declaring host variables
- Determining equivalent SQL data types
- Determining if SQL and host language data types are compatible
- Using indicator variables or host structures, depending on the language
- handling SQL error return codes

For information on reading the syntax diagrams in this chapter, see "How to read the syntax diagrams" on page xvii.

For information on writing embedded SQL application programs in Java, see *DB2 Application Programming Guide and Reference for Java*.

Coding SQL statements in an assembler application

This section helps you with the programming techniques that are unique to coding SQL statements within an assembler program.

Defining the SQL communications area

An assembler program that contains SQL statements must include one or both of the following host variables:
- An SQLCODE variable declared as a fullword integer
- An SQLSTATE variable declared as a character string of length 5 (CL5)

Or,
- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within a BEGIN DECLARE SECTION and END DECLARE SECTION statement in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in an assembler program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

If your program is reentrant, you must include the SQLCA within a unique data area acquired for your task (a DSECT). For example, at the beginning of your program, specify:

```
PROGAREA DSECT
         EXEC SQL INCLUDE SQLCA
```

As an alternative, you can create a separate storage area for the SQLCA and provide addressability to that area.

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:
* CALL...USING DESCRIPTOR *descriptor-name*
* DESCRIBE *statement-name* INTO *descriptor-name*
* DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
* DESCRIBE INPUT *statement-name* INTO *descriptor-name*
* DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
* DESCRIBE TABLE *host-variable* INTO *descriptor-name*
* EXECUTE...USING DESCRIPTOR *descriptor-name*
* FETCH...USING DESCRIPTOR *descriptor-name*
* OPEN...USING DESCRIPTOR *descriptor-name*
* PREPARE...INTO *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. You can code an SQLDA in an assembler program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

You must place SQLDA declarations before the first SQL statement that references the data descriptor unless you use the precompiler option TWOPASS. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLDA fields.

Embedding SQL statements

You can code SQL statements in an assembler program wherever you can use executable statements.

Each SQL statement in an assembler program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in an assembler program as follows:

```
EXEC SQL UPDATE DSN8710.DEPT                          X
            SET MGRNO = :MGRNUM                        X
            WHERE DEPTNO = :INTDEPT
```

Comments: You cannot include assembler comments in SQL statements. However, you can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for assembler statements, except that you must specify EXEC SQL within one line. Any part of the statement that does not fit on one line can appear on subsequent lines, beginning at the continuation margin (column 16, the default). Every line of the statement, except the last, must have a continuation character (a non-blank character) immediately after the right margin in column 72.

Declaring tables and views: Your assembler program should include a DECLARE statement to describe each table and view the program accesses.

Including code: To include SQL statements or assembler host variable declaration statements from a member of a partitioned data set, place the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements.

Margins: The precompiler option MARGINS allows you to set a left margin, a right margin, and a continuation margin. The default values for these margins are columns 1, 71, and 16, respectively. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement. If you use the default margins, you can place an SQL statement anywhere between columns 2 and 71.

Names: You can use any valid assembler name for a host variable. However, do not use external entry names or access plan names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for DB2.

The first character of a host variable used in embedded SQL cannot be an underscore. However, you can use an underscore as the first character in a symbol that is *not* used in embedded SQL.

Statement labels: You can prefix an SQL statement with a label. The first line of an SQL statement can use a label beginning in the left margin (column 1). If you do not use a label, leave column 1 blank.

Assembler

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the assembler source code and must be within the scope of the SQL statements that WHENEVER affects.

Special assembler considerations: The following considerations apply to programs written in assembler:

- To allow for reentrant programs, the precompiler puts all the variables and structures it generates within a DSECT called SQLDSECT, and generates an assembler symbol called SQLDLEN. SQLDLEN contains the length of the DSECT.

  Your program must allocate an area of the size indicated by SQLDLEN, initialize it, and provide addressability to it as the DSECT SQLDSECT.

CICS

An example of code to support reentrant programs, running under CICS, follows:

```
DFHEISTG DSECT
         DFHEISTG
         EXEC SQL INCLUDE SQLCA
*
         DS    0F
SQDWSREG EQU   R7
SQDWSTOR DS    (SQLDLEN)C  RESERVE STORAGE TO BE USED FOR SQLDSECT
  .
  .
  .

&XPROGRM DFHEIENT CODEREG=R12,EIBREG=R11,DATAREG=R13
*
*
*  SQL WORKING STORAGE
         LA    SQDWSREG,SQDWSTOR     GET ADDRESS OF SQLDSECT
         USING SQLDSECT,SQDWSREG     AND TELL ASSEMBLER ABOUT IT
*
```

TSO

The sample program in *prefix*.SDSNSAMP(DSNTIAD) contains an example of how to acquire storage for the SQLDSECT in a program that runs in a TSO environment.

- DB2 does not process set symbols in SQL statements.
- Generated code can include more than two continuations per comment.
- Generated code uses literal constants (for example, =F'-84'), so an LTORG statement might be necessary.
- Generated code uses registers 0, 1, 14, and 15. Register 13 points to a save area that the called program uses. Register 15 does not contain a return code after a call generated by an SQL statement.

> **CICS**
>
> A CICS application program uses the DFHEIENT macro to generate the entry point code. When using this macro, consider the following:
>
> – If you use the default DATAREG in the DFHEIENT macro, register 13 points to the save area.
>
> – If you use any other DATAREG in the DFHEIENT macro, you must provide addressability to a save area.
>
> For example, to use SAVED, you can code instructions to save, load, and restore register 13 around each SQL statement as in the following example.
>
> ```
> ST    13,SAVER13       SAVE REGISTER 13
> LA    13,SAVED         POINT TO SAVE AREA
> EXEC  SQL . . .
> L     13,SAVER13       RESTORE REGISTER 13
> ```

• If you have an addressability error in precompiler-generated code because of input or output host variables in an SQL statement, check to make sure that you have enough base registers.

• Do not put CICS translator options in the assembly source code. Instead, pass the options to the translator by using the PARM field.

Using host variables

You must explicitly declare each host variable before its first use in an SQL statement if you specify the precompiler option ONEPASS. If you specify the precompiler option TWOPASS, you must declare the host variable before its use in the statement DECLARE CURSOR.

You can precede the assembler statements that define host variables with the statement BEGIN DECLARE SECTION, and follow the assembler statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

You can declare host variables in normal assembler style (DC or DS), depending on the data type and the limitations on that data type. You can specify a value on DC or DS declarations (for example, DC H'5'). The DB2 precompiler examines only packed decimal declarations.

A colon (:) must precede all host variables in an SQL statement.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Declaring host variables

Only some of the valid assembler declarations are valid host variable declarations. If the declaration for a host variable is not valid, then any SQL statement that references the variable might result in the message ″UNDECLARED HOST VARIABLE″.

Assembler

Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations. The numeric *value* specifies the scale of the packed decimal variable. If *value* does not include a decimal point, the scale is 0.

For floating point data types (E, EH, EB, D, DH, and DB), DB2 uses the FLOAT precompiler option to determine whether the host variable is in IEEE floating point or System/390 floating point format. If the precompiler option is FLOAT(S390), you need to define your floating point host variables as E, EH, D, or DH. If the precompiler option is FLOAT(IEEE), you need to define your floating point host variables as EB or DB. DB2 converts all floating point input data to System/390 floating point before storing it.
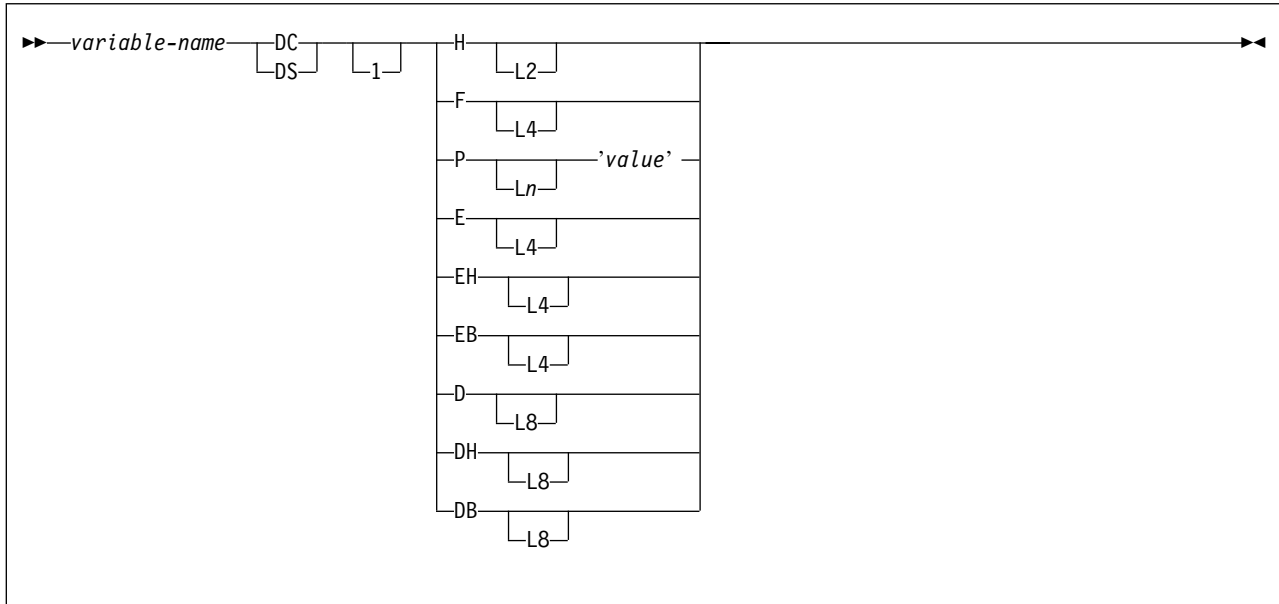
Figure 23. Numeric host variables

Character host variables: There are three valid forms for character host variables:
- Fixed-length strings
- Varying-length strings
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 30 on page 114 for the syntax of CLOBs.
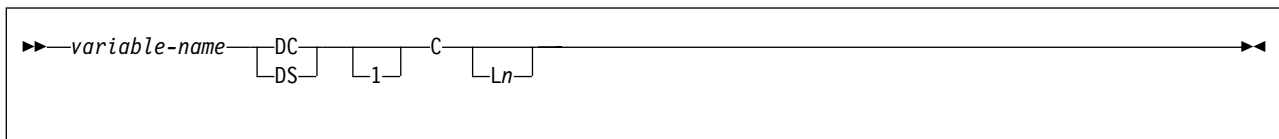
Figure 24. Fixed-length character strings
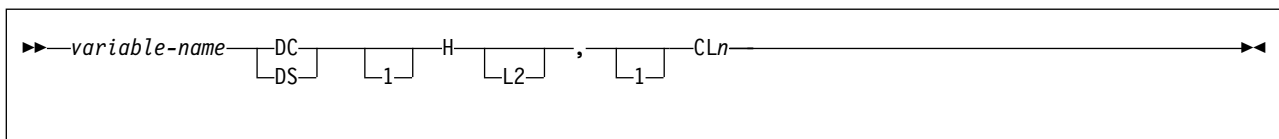
Figure 25. Varying-length character strings

Graphic host variables: There are three valid forms for graphic host variables:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following figures show the syntax for forms other than DBCLOBs. See Figure 30 on page 114 for the syntax of DBCLOBs. In the syntax diagrams, *value* denotes one or more DBCS characters, and the symbols < and > represent shift-out and shift-in characters.
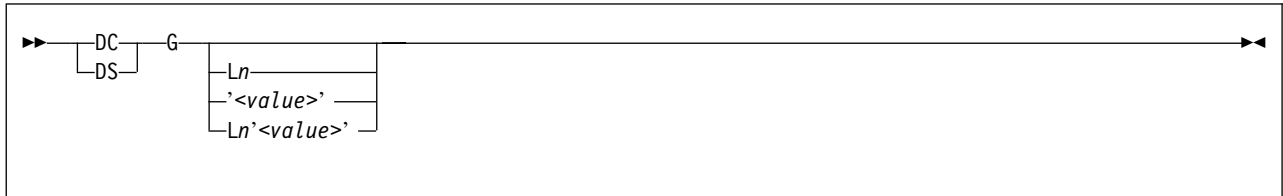
Figure 26. Fixed-length graphic strings
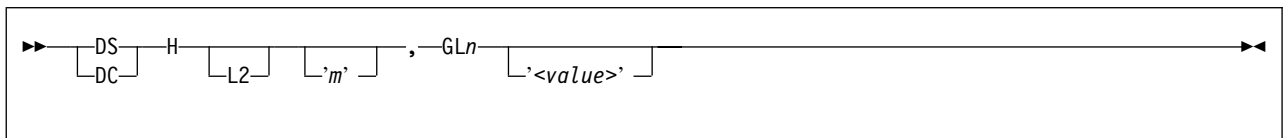
Figure 27. Varying-length graphic strings

Result set locators: The following figure shows the syntax for declarations of result set locators. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for a discussion of how to use these host variables.
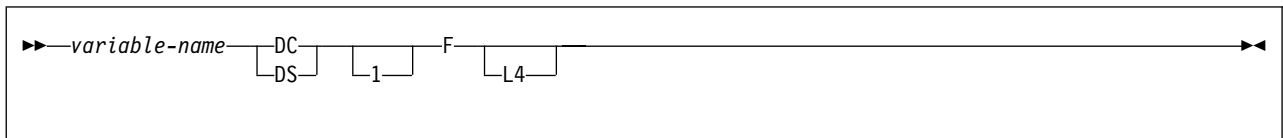
Figure 28. Result set locators

Table Locators: The following figure shows the syntax for declarations of table locators. See "Accessing transition tables in a user-defined function or stored procedure" on page 278 for a discussion of how to use these host variables.
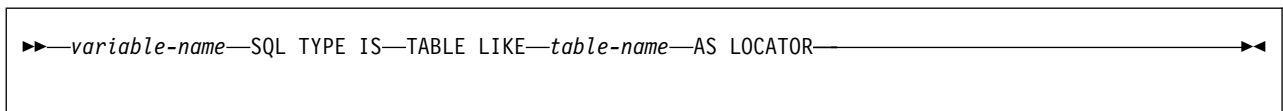
Figure 29. Table locators

LOB variables and locators: The following figure shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators.

If you specify the length of the LOB in terms of KB, MB, or GB, you must leave no spaces between the length and K, M, or B.

See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

Assembler

```
►►──variable-name──SQL──TYPE──IS─┬─────BINARY LARGE OBJECT────┬──length──┬────┬──►◄
                                 ├─BLOB──────────────────────┤          ├─K──┤
                                 ├───CHARACTER LARGE OBJECT───┤          ├─M──┤
                                 ├──CHAR LARGE OBJECT─────────┤          └─G──┘
                                 ├──CLOB──────────────────────┤
                                 ├─DBCLOB─────────────────────┘
                                 ├──BLOB_LOCATOR──────
                                 ├─CLOB_LOCATOR───────
                                 └─DBCLOB_LOCATOR─────
```
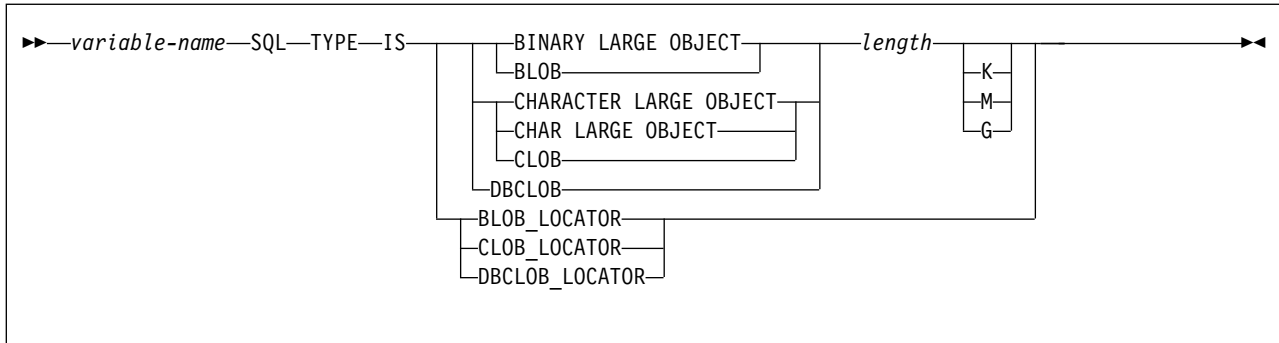
Figure 30. LOB variables and locators

> ***ROWIDs:*** The following figure shows the syntax for declarations of ROWID
> variables. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for
> a discussion of how to use these host variables.

```
►►──variable-name──SQL TYPE IS──ROWID────────────────────────────────────►◄
```
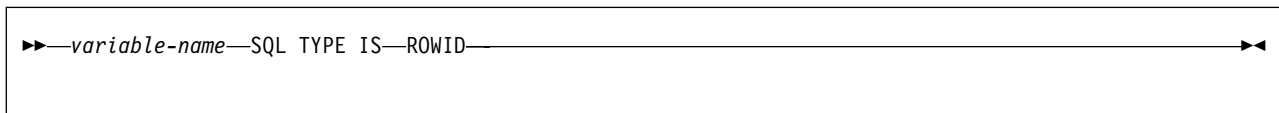
Figure 31. ROWID variables

Determining equivalent SQL and assembler data types

Table 7 describes the SQL data type, and base SQLTYPE and SQLLEN values, that
the precompiler uses for the host variables it finds in SQL statements. If a host
variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE
plus 1.

Table 7. SQL data types the precompiler uses for assembler declarations

Assembler Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
DS HL2	500	2	SMALLINT
DS FL4	496	4	INTEGER
DS P'value' DS PLn'value' or DS PLn 1<=n<=16	484	p in byte 1, s in byte 2	DECIMAL(p,s) See the description for DECIMAL(p,s) in Table 8 on page 115.
DS EL4 DS EHL4 DS EBL4	480	4	REAL or FLOAT (n) 1<=n<=21
DS DL8 DS DHL8 DS DBL8	480	8	DOUBLE PRECISION, or FLOAT (n) 22<=n<=53
DS CLn 1<=n<=255	452	n	CHAR(n)
DS HL2,CLn 1<=n<=255	448	n	VARCHAR(n)
DS HL2,CLn n>255	456	n	VARCHAR(n)
DS GLm 2<=m<=254[1]	468	n	GRAPHIC(n)[2]

Table 7. SQL data types the precompiler uses for assembler declarations (continued)

Assembler Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
DS HL2,GL*m* 2<=*m*<=254[1]	464	*n*	VARGRAPHIC(*n*)[2]
DS HL2,GL*m* *m*>254[1]	472	*n*	VARGRAPHIC(*n*)[2]
DS FL4	972	4	Result set locator[2]
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table locator[2]
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator[2]
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator[3]
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator[3]
SQL TYPE IS BLOB(*n*) 1≤*n*≤2147483647	404	*n*	BLOB(*n*)
SQL TYPE IS CLOB(*n*) 1≤*n*≤2147483647	408	*n*	CLOB(*n*)
SQL TYPE IS DBCLOB(*n*) 1≤*n*≤1073741823[2]	412	*n*	DBCLOB(*n*)[2]
SQL TYPE IS ROWID	904	40	ROWID

Note:

1. *m* is the number of bytes.

2. *n* is the number of double-byte characters.

3. This data type cannot be used as a column type.

Table 8 helps you define host variables that receive output from the database. You can use Table 8 to determine the assembler data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 8 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 8. SQL data types mapped to typical assembler declarations

SQL Data Type	Assembler Equivalent	Notes
SMALLINT	DS HL2	
INTEGER	DS F	

Assembler

Table 8. SQL data types mapped to typical assembler declarations (continued)

SQL Data Type	Assembler Equivalent	Notes
DECIMAL(*p,s*) or NUMERIC(*p,s*)	DS P*'value'* DS PL*n'value'* DS PL*n*	*p* is precision; *s* is scale. 1<=*p*<=31 and 0<=*s*<=*p*. 1<=*n*<=16. *value* is a literal value that includes a decimal point. You must use L*n*, *value*, or both. It is recommended that you use only *value*.
		Precision: If you use L*n*, it is 2*n*-1; otherwise, it is the number of digits in *value*. *Scale:* If you use *value*, it is the number of digits to the right of the decimal point; otherwise, it is 0.
		For efficient use of indexes: Use *value*. If *p* is *even*, do not use L*n* and be sure the precision of *value* is *p* and the scale of *value* is *s*. If *p* is *odd*, you can use L*n* (although it is not advised), but you must choose *n* so that 2*n*-1=*p*, and *value* so that the scale is *s*. Include a decimal point in *value*, even when the scale of *value* is 0.
REAL or FLOAT(*n*)	DS EL4 DS EHL4 DS EBL4[1]	1<=*n*<=21
DOUBLE PRECISION, DOUBLE, or FLOAT(*n*)	DS DL8 DS DHL8 DS DBL8[1]	22<=*n*<=53
CHAR(*n*)	DS CL*n*	1<=*n*<=255
VARCHAR(*n*)	DS HL2,CL*n*	
GRAPHIC(*n*)	DS GL*m*	*m* is expressed in bytes. *n* is the number of double-byte characters. 1<=*n*<=127
VARGRAPHIC(*n*)	DS HL2,GL*x* DS HL2'*m*',GL*x*'<*value*>'	*x* and *m* are expressed in bytes. *n* is the number of double-byte characters. < and > represent shift-out and shift-in characters.
DATE	DS,CL*n*	If you are using a date exit routine, *n* is determined by that routine; otherwise, *n* must be at least 10.
TIME	DS,CL*n*	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	DS,CL*n*	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, truncation occurs on the microseconds part.
Result set locator	DS F	Use this data type only to receive result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.

Table 8. SQL data types mapped to typical assembler declarations (continued)

SQL Data Type	Assembler Equivalent	Notes
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	1≤n≤2147483647
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	1≤n≤2147483647
DBCLOB(*n*)	SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. 1≤n≤1073741823
ROWID	SQL TYPE IS ROWID	

Note:

1. IEEE floating point host variables are not supported in user-defined functions and stored procedures.

Notes on assembler variable declaration and usage

You should be aware of the following when you declare assembler variables.

Host graphic data type: You can use the assembler data type "host graphic" in SQL statements when the precompiler option GRAPHIC is in effect. However, you cannot use assembler DBCS literals in SQL statements, even when GRAPHIC is in effect.

Character host variables: If you declare a host variable as a character string without a length, for example DC C 'ABCD', DB2 interprets it as length 1. To get the correct length, give a length attribute (for example, DC CL4'ABCD').

Floating point host variables: All floating point data is stored in DB2 in System/390 floating point format. However, your host variable data can be in System/390 floating point format or IEEE floating point format. DB2 uses the FLOAT(S390|IEEE) precompiler option to determine whether your floating point host variables are in IEEE floating point format or System/390 floating point format. DB2 does no checking to determine whether the host variable declarations or format of the host variable contents match the precompiler option. Therefore, you need to ensure that your floating point host variable types and contents match the precompiler option.

Special Purpose Assembler Data Types: The locator data types are assembler language data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 278

LOB locators "Chapter 13. Programming for large objects (LOBs)" on page 229

Overflow: Be careful of overflow. For example, suppose you retrieve an INTEGER column value into a DS H host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a host variable declared as DS CL70, the rightmost ten characters of the retrieved string are truncated. If you retrieve a floating-point or decimal column value into a host variable declared as DS F, it removes any fractional part of the value.

Determining compatibility of SQL and assembler data types

Assembler host variables used in SQL statements must be type compatible with the columns with which you intend to use them.

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a numeric assembler host variable.
- Character data types are compatible with each other: A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length assembler character host variable.
- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.
- Graphic data types are compatible with each other: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length assembler graphic character host variable.
- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:
  - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column
  - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
  - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.
  - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.
  - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length assembler character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB host variable.
- The ROWID column is compatible only with a ROWID host variable.
- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. For information on assigning and comparing distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables

An indicator variable is a 2-byte integer (*DS HL2*). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information on indicator variables, see "Using indicator variables with host variables" on page 70 or Chapter 2 of *DB2 SQL Reference*.

Example:

Given the statement:
```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,           X
                               :DAY :DAYIND,      X
                               :BGN :BGNIND,      X
                               :END :ENDIND
```

You can declare variables as follows:
```
CLSCD    DS CL7
DAY      DS HL2
BGN      DS CL8
END      DS CL8
DAYIND   DS HL2             INDICATOR VARIABLE FOR DAY
BGNIND   DS HL2             INDICATOR VARIABLE FOR BGN
ENDIND   DS HL2             INDICATOR VARIABLE FOR END
```

The following figure shows the syntax for a valid indicator variable.
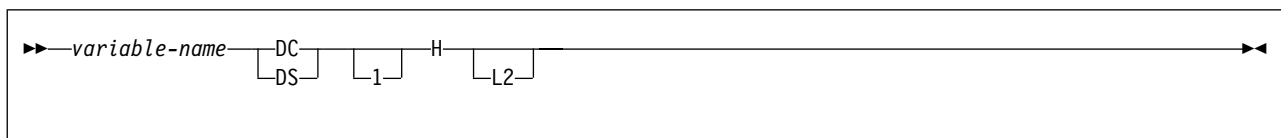
Figure 32. Indicator variable

Handling SQL error return codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL error return codes" on page 76.

DSNTIAR syntax

CALL DSNTIAR,(*sqlca, message, lrecl*),MF=(E,PARM)

The DSNTIAR parameters have the following meanings:

sqlca
> An SQL communication area.

message
> An output area, defined as a varying length string, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.
>
> The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
LINES    EQU   10
LRECL    EQU   132
   .
   .
   .
MESSAGE  DS    H,CL(LINES*LRECL)
         ORG   MESSAGE
MESSAGEL DC    AL2(LINES*LRECL)
MESSAGE1 DS    CL(LRECL)     text line 1
MESSAGE2 DS    CL(LRECL)     text line 2
   .
   .
   .
MESSAGEn DS    CL(LRECL)     text line n
   .
   .
   .
CALL DSNTIAR,(SQLCA, MESSAGE, LRECL),MF=(E,PARM)
```

> where MESSAGE is the name of the message output area, LINES is the number of lines in the message output area, and and LRECL is the length of each line.

lrecl
> A fullword containing the logical record length of output messages, between 72 and 240.

The expression MF=(E,PARM) is an MVS macro parameter that indicates dynamic execution. PARM is the name of a data area that contains a list of pointers to DSNTIAR's call parameters.

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSNTIAD, contained in the library *prefix*.SDSNSAMP. See "Appendix B. Sample applications" on page 829 for instructions on how to access and print the source code for the sample program.

> **CICS**
>
> If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:
>
> ```
> CALL DSNTIAC,(eib,commarea,sqlca,msg,lrecl),MF=(E,PARM)
> ```
>
> DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.
>
> *eib*   EXEC interface block
>
> *commarea*   communication area
>
> For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.
>
> You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see member DSN8FRDO in the data set *prefix*.SDSNSAMP.
>
> The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are also in the data set *prefix*.SDSNSAMP.

Macros for assembler applications

Data set DSN710.SDSNMACS contains all DB2 macros available for use.

Coding SQL statements in a C or a C⁺⁺ application

This section helps you with the programming techniques that are unique to coding SQL statements within a C or C⁺⁺ program. Throughout this book, C is used to represent either C/370™ or C⁺⁺, except where noted otherwise.

Defining the SQL communication area

A C program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as long integer. For example:
    ```
    long SQLCODE;
    ```
- An SQLSTATE variable declared as a character array of length 6. For example:
    ```
    char SQLSTATE[6];
    ```

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variable values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

C

Whether you define SQLCODE or SQLSTATE host variables, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)
When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If you specify STDSQL(NO)
When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a C program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes both a structure definition and a static data area named 'sqlca'. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:
* CALL...USING DESCRIPTOR *descriptor-name*
* DESCRIBE *statement-name* INTO *descriptor-name*
* DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
* DESCRIBE INPUT *statement-name* INTO *descriptor-name*
* DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
* DESCRIBE TABLE *host-variable* INTO *descriptor-name*
* EXECUTE...USING DESCRIPTOR *descriptor-name*
* FETCH...USING DESCRIPTOR *descriptor-name*
* OPEN...USING DESCRIPTOR *descriptor-name*
* PREPARE...INTO *descriptor-name*

Unlike the SQLCA, more than one SQLDA can exist in a program, and an SQLDA can have any valid name. You can code an SQLDA in a C program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

A standard declaration includes only a structure definition with the name 'sqlda'. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLDA fields.

You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the precompiler option TWOPASS. You can place an SQLDA declaration wherever C allows a structure definition. Normal C scoping rules apply.

Embedding SQL statements

You can code SQL statements in a C program wherever you can use executable statements.

Each SQL statement in a C program must begin with EXEC SQL and end with a semi-colon (;). The EXEC and SQL keywords must appear all on one line, but the remainder of the statement can appear on subsequent lines.

Because C is case sensitive, you must use uppercase letters to enter all SQL words. You must also keep the case of host variable names consistent throughout the program. For example, if a host variable name is lowercase in its declaration, it must be lowercase in all SQL statements.

You might code an UPDATE statement in a C program as follows:

```
EXEC SQL
    UPDATE DSN8710.DEPT
    SET MGRNO = :mgr_num
    WHERE DEPTNO = :int_dept;
```

Comments: You can include C comments (/* ... */) within SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can use single-line comments (starting with //) in C language statements, but not in embedded SQL. You cannot nest comments.

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string. You can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL statements: You can use a backslash to continue a character-string constant or delimited identifier on the following line.

Declaring tables and views: Your C program should use the statement DECLARE TABLE to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For details, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

Including code: To include SQL statements or C host variable declarations from a member of a partitioned data set, add the following SQL statement in the source code where you want to embed the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use C `#include` statements to include SQL statements or C host variable declarations.

Margins: Code SQL statements in columns 1 through 72, unless you specify other margins to the DB2 precompiler. If EXEC SQL is not within the specified margins, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid C name for a host variable, subject to the following restrictions:

- Do not use DBCS characters.

- Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL' (in any combination of uppercase or lowercase letters). These names are reserved for DB2.

Nulls and NULs: C and SQL differ in the way they use the word null. The C language has a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon). The C NUL is a single character which compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a (nonnull) value. In this chapter, NUL is the null character in C and NULL is the SQL null value.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement labels: You can precede SQL statements with a label, if you wish.

Trigraphs: Some characters from the C character set are not available on all keyboards. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*. The trigraphs that DB2 supports are the same as those that the C/370 compiler supports.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be within the scope of any SQL statements that the statement WHENEVER affects.

Special C considerations:
- Use of the C/370 multi-tasking facility, where multiple tasks execute SQL statements, causes unpredictable results.
- You must run the DB2 precompiler before running the C preprocessor.
- The DB2 precompiler does not support C preprocessor directives.
- If you use conditional compiler directives that contain C code, either place them after the first C token in your application program, or include them in the C program using the `#include` preprocessor directive.

Please refer to the appropriate C documentation for further information on C preprocessor directives.

Using host variables

You must explicitly declare each host variable before its first use in an SQL statement if you specify the ONEPASS precompiler option. If you use the precompiler option TWOPASS, you must declare each host variable before its first use in the statement DECLARE CURSOR.

Precede C statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the C statements with the statement END DECLARE SECTION. You can have more than one host variable declaration section in your program.

A colon (:) must precede all host variables in an SQL statement.

The names of host variables must be unique within the program, even if the host variables are in different blocks, classes, or procedures. You can qualify the host variable names with a structure name to make them unique.

An SQL statement that uses a host variable must be within the scope of that variable.

Host variables must be scalar variables or host structures; they cannot be elements of vectors or arrays (subscripted variables) unless you use the character arrays to hold strings. You can use an array of indicator variables when you associate the array with a host structure.

Declaring host variables

Only some of the valid C declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message ″UNDECLARED HOST VARIABLE″.

Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations.
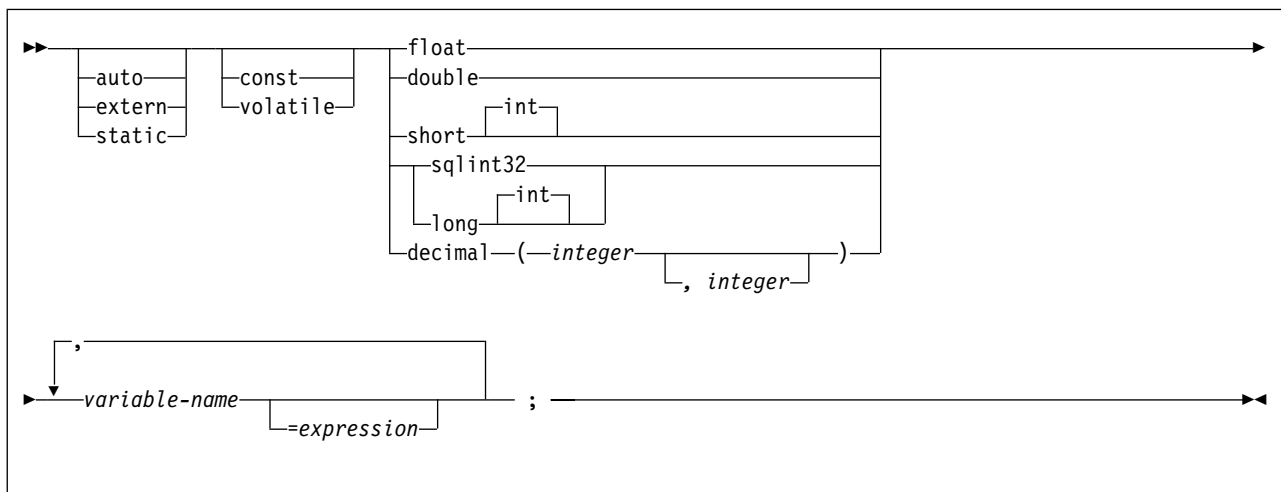
Figure 33. Numeric host variables

Character host variables: There are four valid forms for character host variables:
* Single-character form
* NUL-terminated character form
* VARCHAR structured form
* CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 42 on page 129 for the syntax of CLOBs.
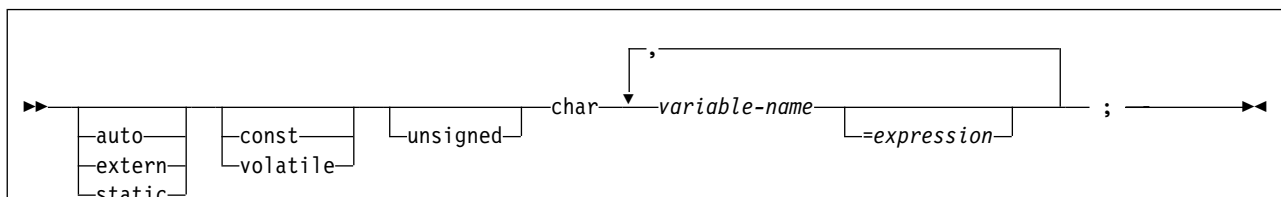
Figure 34. Single-character form

C
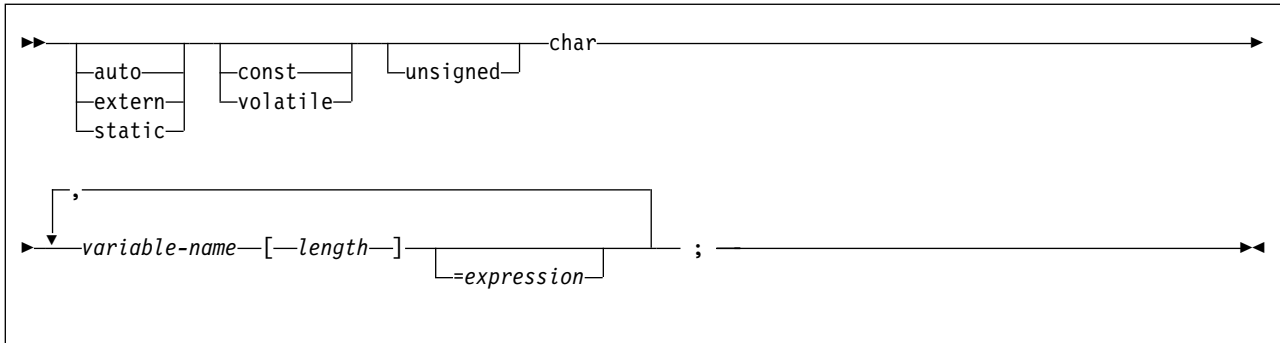
Figure 35. NUL-terminated character form

> ### Notes:
> 1. On input, the string contained by the variable must be NUL-terminated.
> 2. On output, the string is NUL-terminated.
> 3. A NUL-terminated character host variable maps to a varying length character string (except for the NUL).
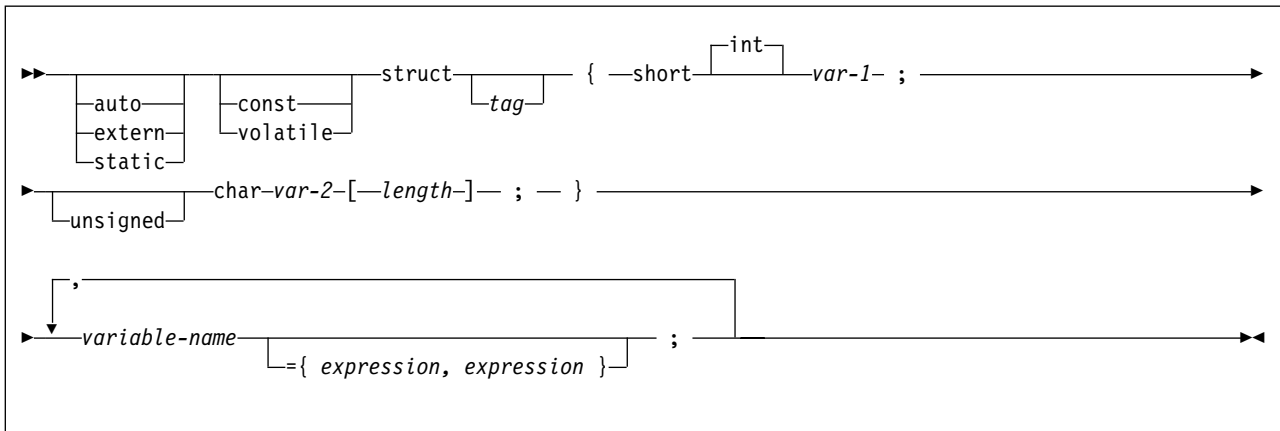
Figure 36. VARCHAR structured form

> ### Notes:
> - *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
> - You can use the struct tag to define other data areas, which you cannot use as host variables.
>
> ### Example:
> ```
> EXEC SQL BEGIN DECLARE SECTION;
>
>     /* valid declaration of host variable vstring */
>
>     struct VARCHAR {
>        short len;
>        char s[10];
>        } vstring;
>
>     /* invalid declaration of host variable wstring */
>
>     struct VARCHAR wstring;
> ```
>
> **Graphic host variables:** There are four valid forms for graphic host variables:

- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form.
- DBCLOBs

You can use the C data type wchar_t to define a host variable that inserts, updates, deletes, and selects data from GRAPHIC or VARGRAPHIC columns.

The following figures show the syntax for forms other than DBCLOBs. See Figure 42 on page 129 for the syntax of DBCLOBs.
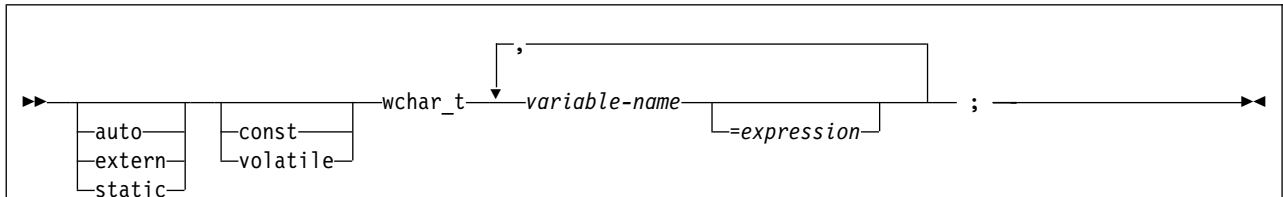
Figure 37. Single-graphic form

The single-graphic form declares a fixed-length graphic string of length 1. You cannot use array notation in *variable-name*.
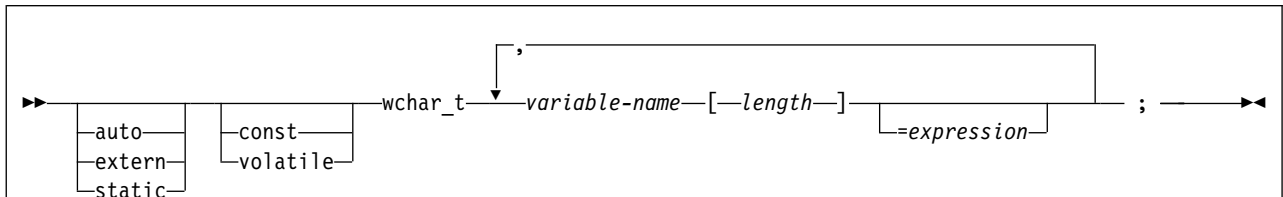
Figure 38. Nul-terminated graphic form

Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. On input, the string in *variable-name* must be NUL-terminated.
3. On output, the string is NUL-terminated.
4. The NUL-terminated graphic form does not accept single byte characters into *variable-name*.
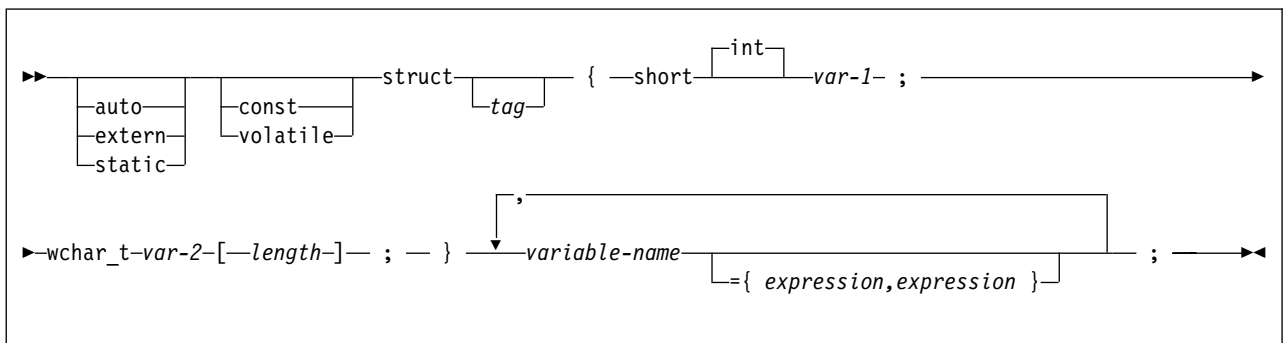
Figure 39. VARGRAPHIC structured form

Notes:

- *length* must be a decimal integer constant greater than 1 and not greater than 16352.
- *var-1* must be less than or equal to *length*.
- *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
- You can use the struct tag to define other data areas, which you cannot use as host variables.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;

    /* valid declaration of host variable vgraph */

    struct VARGRAPH {
        short len;
        wchar_t d[10];
        } vgraph;

    /* invalid declaration of host variable wgraph */

    struct VARGRAPH wgraph;
```

Result set locators: The following figure shows the syntax for declarations of result set locators. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for a discussion of how to use these host variables.
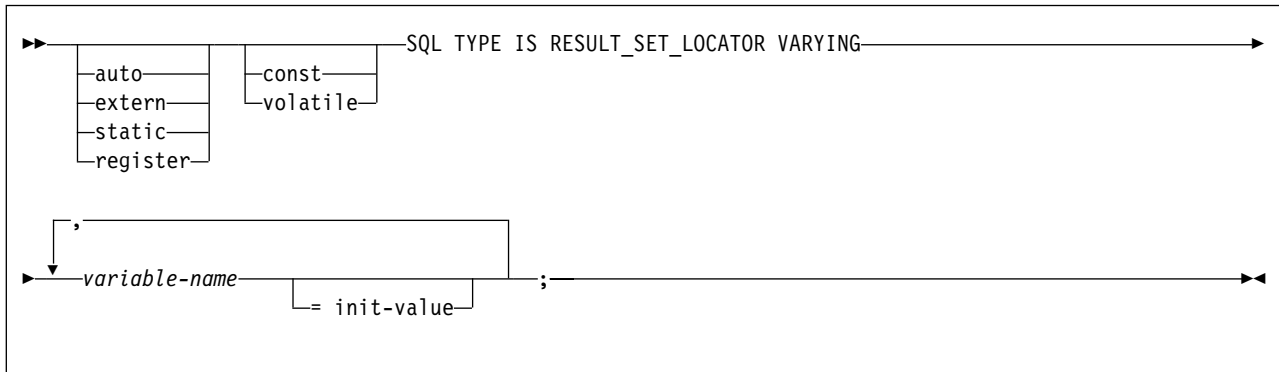
Figure 40. Result set locators

Table Locators: The following figure shows the syntax for declarations of table locators. See "Accessing transition tables in a user-defined function or stored procedure" on page 278 for a discussion of how to use these host variables.

```
►►─┬─────────┬─┬─────────┬──SQL TYPE IS─TABLE LIKE─table-name─AS LOCATOR──────────────►
   ├─auto────┤ ├─const───┤
   ├─extern──┤ └─volatile┘
   ├─static──┤
   └─register┘


     ┌─,───────────────────┐
     ▼                     │
►─────variable-name─┬────────────┬─;──────────────────────────────────────────────►◄
                    └─init-value─┘
```
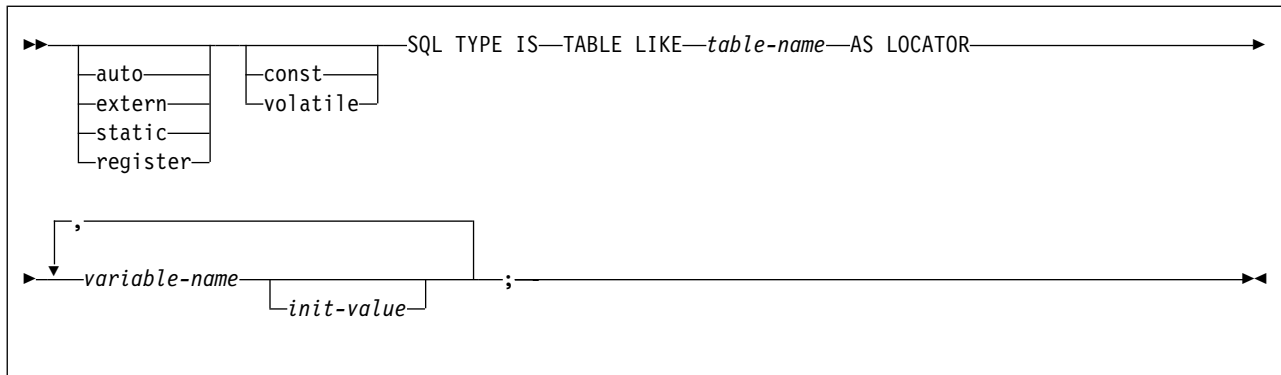
Figure 41. Table locators

LOB Variables and Locators: The following figure shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►─┬─────────┬─┬─────────┬──SQL─TYPE─IS────────────────────────────────────────────►
   ├─auto────┤ ├─const───┤
   ├─extern──┤ └─volatile┘
   ├─static──┤
   └─register┘

►─┬─┬─BINARY LARGE OBJECT──────┬──(─length─┬───┬─)─┬──────────────────────────────►
  │ ├─BLOB─────────────────────┤           ├─K─┤   │
  │ ├─CHARACTER LARGE OBJECT───┤           ├─M─┤   │
  │ ├─CHAR LARGE OBJECT────────┤           └─G─┘   │
  │ ├─CLOB─────────────────────┤                   │
  │ └─DBCLOB────────────────────                    │
  ├─BLOB_LOCATOR───────────────┤                    │
  ├─CLOB_LOCATOR───────────────┤                    │
  └─DBCLOB_LOCATOR─────────────┘

     ┌─,───────────────────┐
     ▼                     │
►─────variable-name─┬────────────┬─;──────────────────────────────────────────────►◄
                    └─init-value─┘
```
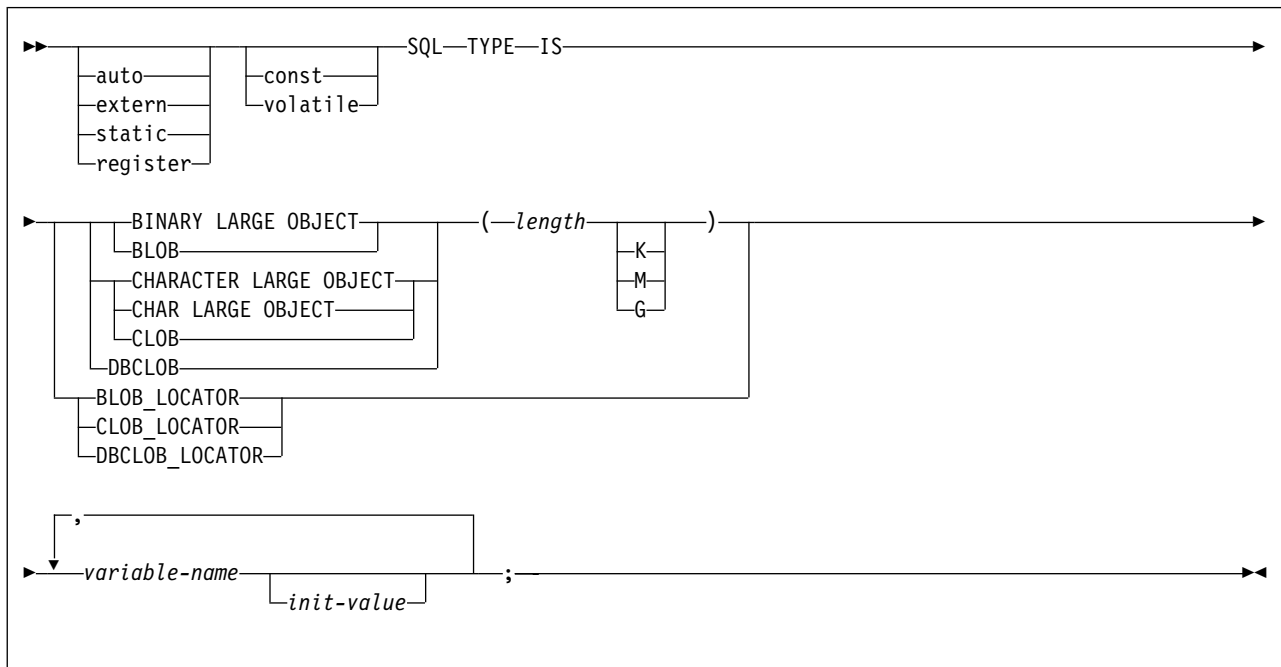
Figure 42. LOB variables and locators

ROWIDs: The following figure shows the syntax for declarations of ROWID variables. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►─┬─────────┬─┬─────────┬──variable-name─SQL TYPE IS─ROWID─;─────────────────────►◄
   ├─auto────┤ ├─const───┤
   ├─extern──┤ └─volatile┘
   ├─static──┤
   └─register┘
```
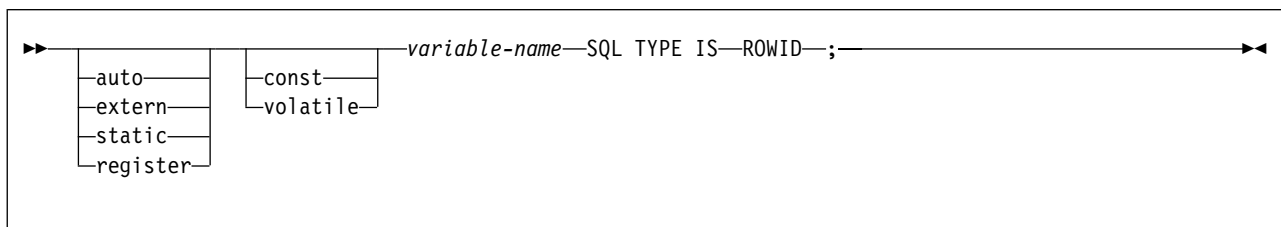
Figure 43. ROWID variables

C
Using host structures

A C host structure contains an ordered group of data fields. For example:

```
struct {char  c1[3];
        struct {short len;
                 char  data[5];
                }c2;
        char  c3[2];
       }target;
```

In this example, *target* is the name of a host structure consisting of the *c1, c2,* and *c3* fields. *c1* and *c3* are character arrays, and *c2* is the host variable equivalent to the SQL VARCHAR data type. The target host structure can be part of another host structure but must be the deepest level of the nested structure.

The following figure shows the syntax for valid host structures.
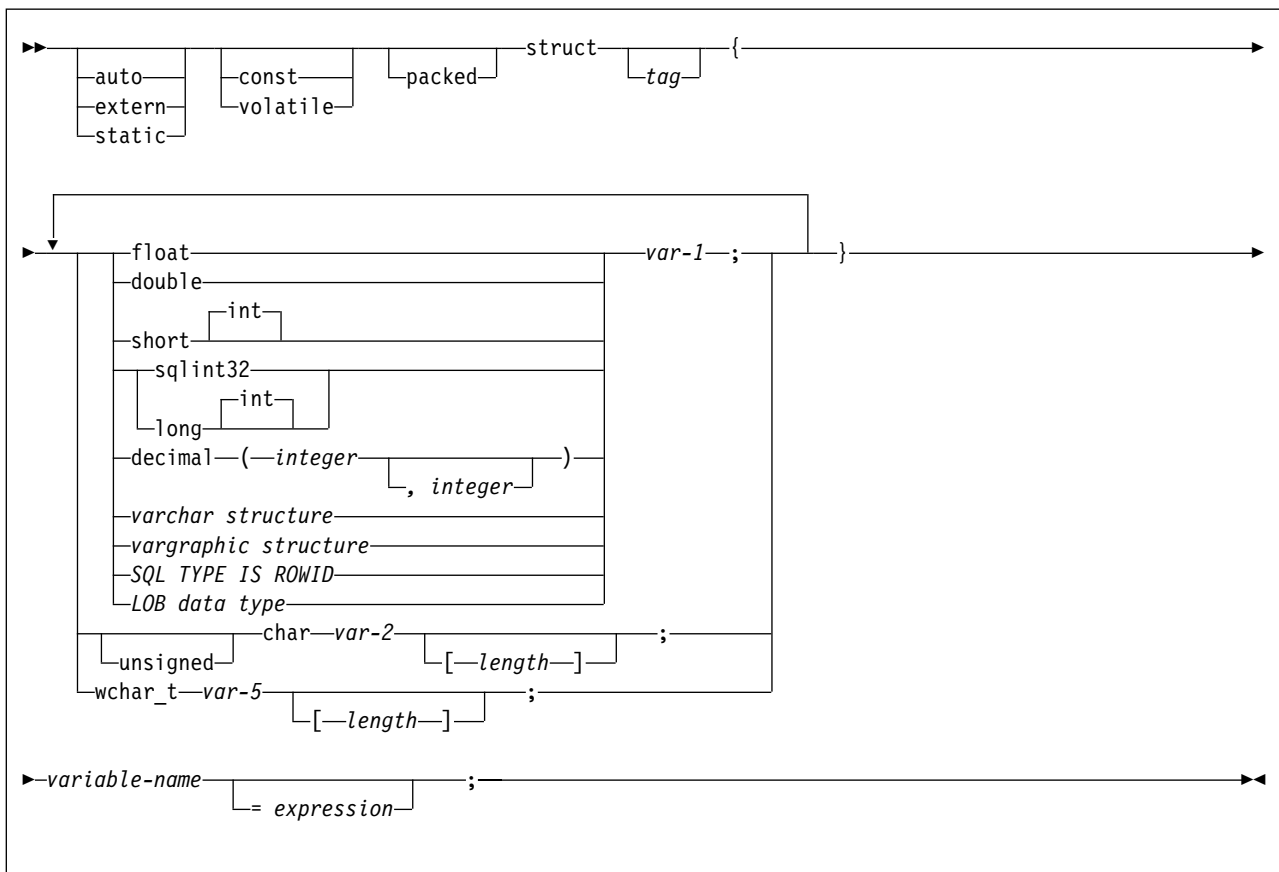
Figure 44. Host structures
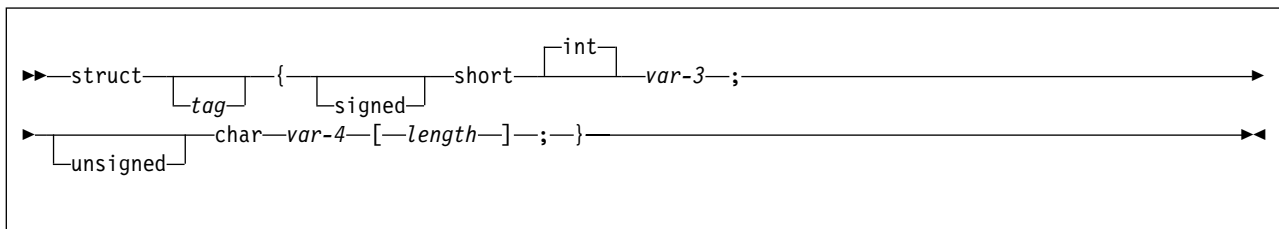
Figure 45. varchar-structure
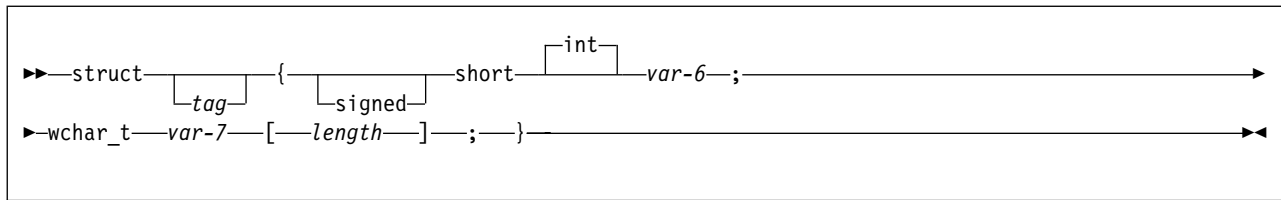
Figure 46. VARGRAPHIC-structure
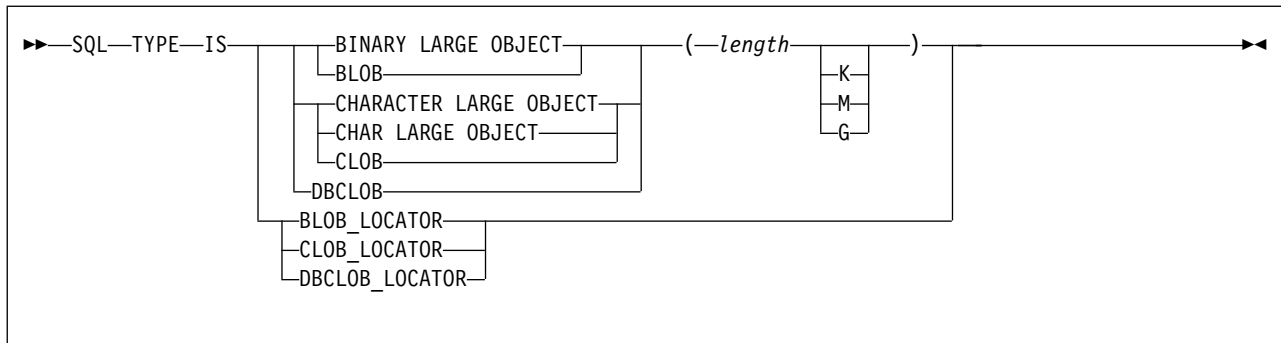
Figure 47. LOB data type

Determining equivalent SQL and C data types

Table 9 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 9. SQL data types the precompiler uses for C declarations

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
short int	500	2	SMALLINT
long int	496	4	INTEGER
decimal(p,s)[1]	484	p in byte 1, s in byte 2	DECIMAL(p,s)[1]
float	480	4	FLOAT (single precision)
double	480	8	FLOAT (double precision)
Single-character form	452	1	CHAR(1)
NUL-terminated character form	460	n	VARCHAR (n-1)
VARCHAR structured form 1<=n<=255	448	n	VARCHAR(n)
VARCHAR structured form n>255	456	n	VARCHAR(n)
Single-graphic form	468	1	GRAPHIC(1)

Table 9. SQL data types the precompiler uses for C declarations (continued)

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
NUL-terminated graphic form (wchar_t)	400	n	VARGRAPHIC (n-1)
VARGRAPHIC structured form 1<=n<128	464	n	VARGRAPHIC(n)
VARGRAPHIC structured form n>127	472	n	VARGRAPHIC(n)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator[2]
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table locator[2]
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator[2]
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator[2]
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator[2]
SQL TYPE IS BLOB(n) 1≤n≤2147483647	404	n	BLOB(n)
SQL TYPE IS CLOB(n) 1≤n≤2147483647	408	n	CLOB(n)
SQL TYPE IS DBCLOB(n) 1≤n≤1073741823	412	n	DBCLOB(n)[3]
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. *p* is the *precision* in SQL terminology which is the total number of digits. In C this is called the *size*.

   s is the *scale* in SQL terminology which is the number of digits to the right of the decimal point. In C, this is called the *precision*.

2. Do not use this data type as a column type.

3. *n* is the number of double-byte characters.

Table 10 on page 133 helps you define host variables that receive output from the database. You can use the table to determine the C data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 10 on page 133 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do

assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 10. SQL data types mapped to typical C declarations

SQL Data Type	C Data Type	Notes
SMALLINT	short int	
INTEGER	long int	
DECIMAL(*p,s*) or NUMERIC(*p,s*)	decimal	You can use the double data type if your C compiler does not have a decimal data type; however, double is not an exact equivalent.
REAL or FLOAT(*n*)	float	1<=*n*<=21
DOUBLE PRECISION or FLOAT(*n*)	double	22<=*n*<=53
CHAR(1)	single-character form	
CHAR(*n*)	no exact equivalent	If *n*>1, use NUL-terminated character form
VARCHAR(*n*)	NUL-terminated character form	If data can contain character NULs (\0), use VARCHAR structured form. Allow at least *n*+1 to accommodate the NUL-terminator.
	VARCHAR structured form	
GRAPHIC(1)	single-graphic form	
GRAPHIC(*n*)	no exact equivalent	If *n*>1, use NUL-terminated graphic form. *n* is the number of double-byte characters.
VARGRAPHIC(*n*)	NUL-terminated graphic form	If data can contain graphic NUL values (\0\0), use VARGRAPHIC structured form. Allow at least *n*+1 to accommodate the NUL-terminator. *n* is the number of double-byte characters.
	VARGRAPHIC structured form	*n* is the number of double-byte characters.
DATE	NUL-terminated character form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 11 characters to accommodate the NUL-terminator.
	VARCHAR structured form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 10 characters.
TIME	NUL-terminated character form	If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 7; to include seconds, the length must be at least 9 to accommodate the NUL-terminator.
	VARCHAR structured form	If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 6; to include seconds, the length must be at least 8.

Table 10. SQL data types mapped to typical C declarations (continued)

SQL Data Type	C Data Type	Notes
TIMESTAMP	NUL-terminated character form	The length must be at least 20. To include microseconds, the length must be 27. If the length is less than 27, truncation occurs on the microseconds part.
	VARCHAR structured form	The length must be at least 19. To include microseconds, the length must be 26. If the length is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	$1 \leq n \leq 2147483647$
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	$1 \leq n \leq 2147483647$
DBCLOB(*n*	SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. $1 \leq n \leq 1073741823$
ROWID	SQL TYPE IS ROWID	

Notes on C variable declaration and usage
You should be aware of the following when you declare C variables.

C data types with no SQL equivalent: C supports some data types and storage classes with no SQL equivalents, for example, register storage class, typedef, and the pointer.

SQL data types with no C equivalent: If your C compiler does not have a decimal data type, then there is no exact equivalent for the SQL DECIMAL data type. In this case, to hold the value of such a variable, you can use:
- An integer or floating-point variable, which converts the value. If you choose integer, you will lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer, or if you want to preserve a fractional value, you can use floating-point numbers. Floating-point numbers are approximations of real numbers. Hence, when you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to get a string representation of a decimal number.

- The DECIMAL function to explicitly convert a value to a decimal data type, as in this example:

```
long duration=10100;  /* 1 year and 1 month */
char result_dt[11];

EXEC SQL SELECT START_DATE + DECIMAL(:duration,8,0)
        INTO :result_dt FROM TABLE1;
```

Floating point host variables: All floating point data is stored in DB2 in System/390 floating point format. However, your host variable data can be in System/390 floating point format or IEEE floating point format. DB2 uses the FLOAT(S390|IEEE) precompiler option to determine whether your floating point host variables are in IEEE floating point or System/390 floating point format. DB2 does no checking to determine whether the contents of a host variable match the precompiler option. Therefore, you need to ensure that your floating point data format matches the precompiler option.

Graphic host variables in user-defined function: The SQLUDF file, which is in data set DSN710.SDSNC.H, contains many data declarations for C language user-defined functions. SQLUDF contains the typedef sqldbchar, which you can use instead of wchar_t. Using sqldbchar lets you manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in DB2. sqldbchar also makes applications easier to port to other DB2 platforms.

Special Purpose C Data Types: The locator data types are C data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator
> "Chapter 24. Using stored procedures for client/server processing" on page 525

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 278

LOB locators "Chapter 13. Programming for large objects (LOBs)" on page 229

String host variables:

If you assign a string of length *n* to a NUL-terminated variable with a length that is:

- less than or equal to *n*, then DB2 inserts the characters into the host variable as long as the characters fit up to length (*n*-1) and appends a NUL at the end of the string. DB2 sets SQLWARN[1] to *W* and any indicator variable you provide to the original length of the source string.
- equal to *n*+1, then DB2 inserts the characters into the host variable and appends a NUL at the end of the string.
- greater than *n*+1, then the rules depend on whether the source string is a value of a fixed-length string column or a varying-length string column. See Chapter 2 of *DB2 SQL Reference* for more information.

PREPARE or DESCRIBE statements: You cannot use a host variable that is of the NUL-terminated form in either a PREPARE or DESCRIBE statement.

L-literals: DB2 tolerates L-literals in C application programs. DB2 allows properly-formed L-literals, although it does not check for all the restrictions that the C compiler imposes on the L-literal. You can use DB2 graphic string constants in SQL statements to work with the L-literal. Do not use L-literals in SQL statements.

Overflow: Be careful of overflow. For example, suppose you retrieve an INTEGER column value into a short integer host variable and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.

Truncation: Be careful of truncation. Ensure the host variable you declare can contain the data and a NUL terminator, if needed. Retrieving a floating-point or decimal column value into a long integer host variable removes any fractional part of the value.

Notes on syntax differences for constants

You should be aware of the following syntax differences for constants.

Decimal constants versus REAL (floating) constants: In C, a string of digits with a decimal point is interpreted as a real constant. In an SQL statement, such a string is interpreted as a decimal constant. You must use exponential notation when specifying a real (that is, floating-point) constant in an SQL statement.

In C, a real (floating-point) constant can have a suffix of f or F to show a data type of *float* or a suffix of l or L to show a type of *long double*. A floating-point constant in an SQL statement must not use these suffixes.

Integer constants: In C, you can provide integer constants in hexadecimal if the first two characters are 0x or 0X. You cannot use this form in an SQL statement.

In C, an integer constant can have a suffix of u or U to show that it is an unsigned integer. An integer constant can have a suffix of l or L to show a long integer. You cannot use these suffixes in SQL statements.

Character and string constants: In C, character constants and string constants can use escape sequences. You cannot use the escape sequences in SQL statements. Apostrophes and quotes have different meanings in C and SQL. In C, you can use quotes to delimit string constants, and apostrophes to delimit character constants. The following examples illustrate the use of quotes and apostrophes in C.

Quotes

```
printf(  "%d lines read. \n", num_lines);
```

Apostrophes

```
#define NUL '\0'
```

In SQL, you can use quotes to delimit identifiers and apostrophes to delimit string constants. The following examples illustrate the use of apostrophes and quotes in SQL.

Quotes

```
SELECT "COL#1" FROM TBL1;
```

Apostrophes

```
SELECT COL1 FROM TBL1 WHERE COL2 = 'BELL';
```

Character data in SQL is distinct from integer data. Character data in C is a subtype of integer data.

Determining compatibility of SQL and C data types

C host variables used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with any C host variable defined as type short int, long int, decimal, float, or double.
- Character data types are compatible with each other: A CHAR, VARCHAR, or CLOB column is compatible with a single character, NUL-terminated, or VARCHAR structured form of a C character host variable.
- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.
- Graphic data types are compatible with each other. A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a single character, NUL-terminated, or VARGRAPHIC structured form of a C graphic host variable.
- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:
  - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column
  - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
  - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.
  - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.
  - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB host variable.
- The ROWID column is compatible only with a ROWID host variable.
- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. For information on assigning and comparing distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Varying-length strings: *For varying-length BIT data,* use the VARCHAR structured form. Some C string manipulation functions process NUL-terminated strings and

others process strings that are not NUL-terminated. The C string manipulation functions that process NUL-terminated strings cannot handle bit data; the functions might misinterpret a NUL character to be a NUL-terminator.

Using indicator variables

An indicator variable is a 2-byte integer (short int). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see "Using indicator variables with host variables" on page 70.

Example:

Given the statement:
```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                               :Day :DayInd,
                               :Bgn :BgnInd,
                               :End :EndInd;
```

You can declare variables as follows:
```
EXEC SQL BEGIN DECLARE SECTION;
char  ClsCd[8];
char  Bgn[9];
char  End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;
```

The following figure shows the syntax for a valid indicator variable.
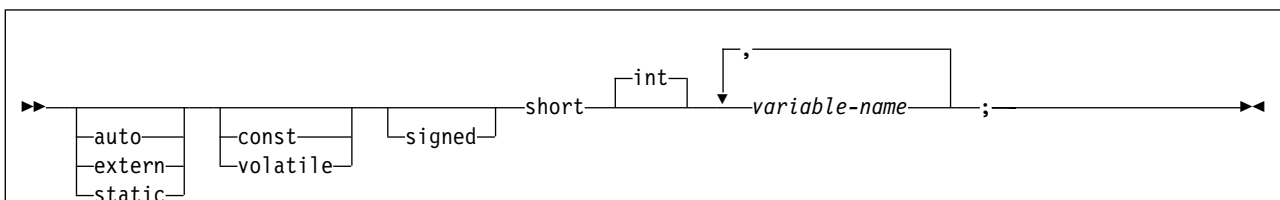
Figure 48. Indicator variable

The following figure shows the syntax for a valid indicator array.
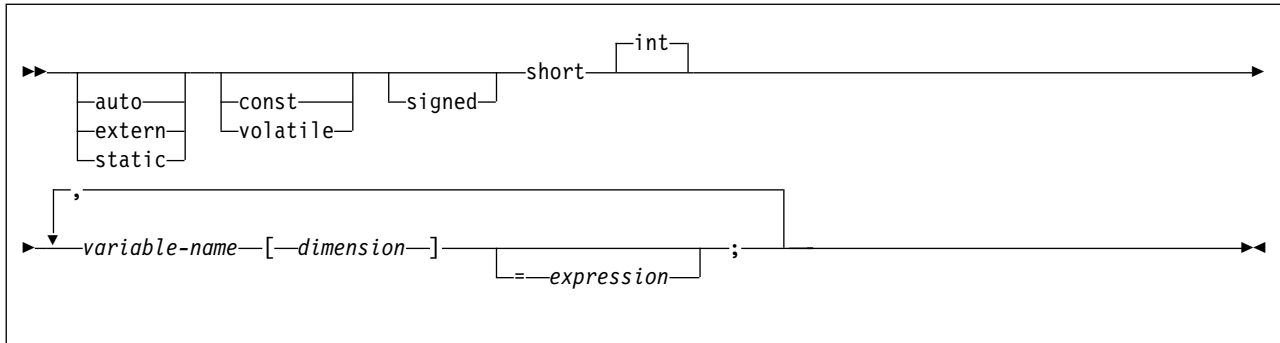
Figure 49. Host structure indicator array

Note:

Dimension must be an integer constant between 1 and 32767.

Handling SQL error return codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL error return codes" on page 76.

DSNTIAR syntax

rc = dsntiar(*&sqlca, &message, &lrecl*);

The DSNTIAR parameters have the following meanings:

&sqlca
> An SQL communication area.

&message
> An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.
>
> The output lines of text, each line being the length specified in *&lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
#define data_len 132
#define data_dim 10
struct error_struct {
   short int error_len;
   char error_text[data_dim][data_len];
   } error_message = {data_dim * data_len};
 :
rc = dsntiar(&sqlca, &error_message, &data_len);
```

> where error_message is the name of the message output area, data_dim is the number of lines in the message output area, and data_len is length of each line.

&lrecl
> A fullword containing the logical record length of output messages, between 72 and 240.

To inform your compiler that DSNTIAR is an assembler language program, include one of the following statements in your application.

For C, include:

```
#pragma linkage (dsntiar,OS)
```

For C⁺⁺, include a statement similar to this:

```
extern "OS" short int dsntiar(struct sqlca *sqlca,
                              struct error_struct *error_message,
                              int *data_len);
```

Examples of calling DSNTIAR from an application appear in the DB2 sample C program DSN8BD3 and in the sample C⁺⁺ program DSN8BE3. Both are in the library DSN8710.SDSNSAMP. See "Appendix B. Sample applications" on page 829 for instructions on how to access and print the source code for the sample program.

CICS

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
rc = DSNTIAC(&eib, &commarea, &sqlca, &message, &lrecl);
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

&eib   EXEC interface block

&commarea
          communication area

For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Considerations for C⁺⁺

When you code SQL in a C⁺⁺ program, be aware of the following:

Using C⁺⁺ data types as host variables: You can use class members as host variables. Class members used as host variables are accessible to any SQL statement within the class.

You cannot use class objects as host variables.

Coding SQL statements in a COBOL application

This section helps you with the programming techniques that are unique to coding SQL statements within a COBOL program.

Except where noted otherwise, this information pertains to all COBOL compilers supported by DB2 for OS/390 and z/OS.

Defining the SQL communication area

A COBOL program that contains SQL statements must include one or both of the following host variables:
- An SQLCODE variable declared as PIC S9(9) BINARY, PIC S9(9) COMP-4, PIC S9(9) COMP-5, or PICTURE S9(9) COMP
- An SQLSTATE variable declared as PICTURE X(5)

Or,
- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)
When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

When you use the precompiler option STDSQL(YES), you must declare an SQLCODE variable. DB2 declares an SQLCA area for you in the WORKING-STORAGE SECTION. DB2 controls that SQLCA, so your application programs should not make assumptions about its structure or location.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If you specify STDSQL(NO)
When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a COBOL program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

You can specify INCLUDE SQLCA or a declaration for SQLCODE wherever you can specify a 77 level or a record description entry in the WORKING-STORAGE SECTION. You can declare a stand-alone SQLCODE variable in either the WORKING-STORAGE SECTION or LINKAGE SECTION.

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:
* CALL...USING DESCRIPTOR *descriptor-name*
* DESCRIBE *statement-name* INTO *descriptor-name*
* DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
* DESCRIBE INPUT *statement-name* INTO *descriptor-name*
* DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
* DESCRIBE TABLE *host-variable* INTO *descriptor-name*
* EXECUTE...USING DESCRIPTOR *descriptor-name*
* FETCH...USING DESCRIPTOR *descriptor-name*
* OPEN...USING DESCRIPTOR *descriptor-name*
* PREPARE...INTO *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. The DB2 SQL INCLUDE statement does not provide an SQLDA mapping for COBOL. You can define the SQLDA using one of the following two methods:

* For COBOL programs compiled with any compiler *except* the OS/VS COBOL compiler, you can code the SQLDA declarations in your program. For more information, see "Using dynamic SQL in COBOL" on page 524. You must place SQLDA declarations in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program, wherever you can specify a record description entry in that section.

* For COBOL programs compiled with any COBOL compiler, you can call a subroutine (written in C, PL/I, or assembler language) that uses the DB2 INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions you need. You must use this method if you compile your program using OS/VS COBOL. The SQLDA definition includes the POINTER data type, which OS/VS COBOL does not support. For more information on using dynamic SQL, see "Chapter 23. Coding dynamic SQL in application programs" on page 495.

You must place SQLDA declarations before the first SQL statement that references the data descriptor. An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Embedding SQL statements

You can code SQL statements in the COBOL program sections shown in Table 11.

Table 11. Allowable SQL statements for COBOL program sections

SQL Statement	Program Section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE SQLCA	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE text-file-name	PROCEDURE DIVISION or DATA DIVISION[1]
DECLARE TABLE DECLARE CURSOR	DATA DIVISION or PROCEDURE DIVISION
Other	PROCEDURE DIVISION

Table 11. Allowable SQL statements for COBOL program sections (continued)

SQL Statement	Program Section

Note: [1] When including host variable declarations, the INCLUDE statement must be in the WORKING-STORAGE SECTION or the LINKAGE SECTION.

You cannot put SQL statements in the DECLARATIVES section of a COBOL program.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If the SQL statement appears between two COBOL statements, the period is optional and might not be appropriate. If the statement appears in an IF...THEN set of COBOL statements, leave off the ending period to avoid inadvertently ending the IF statement. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a COBOL program as follows:

```
EXEC SQL
    UPDATE DSN8710.DEPT
    SET MGRNO = :MGR-NUM
    WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments: You can include COBOL comment lines (* in column 7) in SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. The precompiler also treats COBOL debugging and page eject lines (D or / in column 7) as comment lines. For an SQL INCLUDE statement, DB2 treats any text that follows the period after END-EXEC and is on the same line as END-EXEC as a comment.

In addition, you can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL statements: The rules for continuing a character string constant from one line to the next in an SQL statement embedded in a COBOL program are the same as those for continuing a non-numeric literal in COBOL. However, you can use either a quotation mark or an apostrophe as the first nonblank character in area B of the continuation line. The same rule applies for the continuation of delimited identifiers and does not depend on the string delimiter option.

To conform with SQL standard, delimit a character string constant with an apostrophe, and use a quotation mark as the first nonblank character in area B of the continuation line for a character string constant.

Declaring tables and views: Your COBOL program should include the statement DECLARE TABLE to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. You should include the DCLGEN members in the DATA DIVISION. For details, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

Dynamic SQL in a COBOL program: In general, COBOL programs can easily handle dynamic SQL statements. COBOL programs can handle SELECT statements if the data types and the number of fields returned are fixed. If you want

to use variable-list SELECT statements, use an SQLDA. See "Defining SQL descriptor areas" on page 142 for more information on SQLDA.

Including code: To include SQL statements or COBOL host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name END-EXEC.
```

You cannot nest SQL INCLUDE statements. Do not use COBOL verbs to include SQL statements or COBOL host variable declarations, or use the SQL INCLUDE statement to include CICS preprocessor related code. In general, use the SQL INCLUDE only for SQL-related coding.

Margins: Code SQL statements in columns 12 through 72. If EXEC SQL starts before column 12, the DB2 precompiler does not recognize the SQL statement.

The precompiler option MARGINS allows you to set new left and right margins between 1 and 80. However, you must not code the statement EXEC SQL before column 12.

Names: You can use any valid COBOL name for a host variable. Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement labels: You can precede executable SQL statements in the PROCEDURE DIVISION with a paragraph name, if you wish.

WHENEVER statement: The target for the GOTO clause in an SQL statement WHENEVER must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

Special COBOL considerations: The following considerations apply to programs written in COBOL:
- In a COBOL program that uses elements in a multi-level structure as host variable names, the DB2 precompiler generates the lowest two-level names. If you then compile the COBOL program using OS/VS COBOL, the compiler issues messages IKF3002I and IKF3004I. If you compile the program using VS COBOL II or later compilers, you can eliminate these messages.
- Use of the COBOL compiler options DYNAM and NODYNAM depends on the operating environment.

```
┌─ TSO and IMS ─────────────────────────────────────────────────┐
│                                                                 │
│  You can specify the option DYNAM when compiling a COBOL        │
│  program if you use VS COBOL II or COBOL/370™, or if you use    │
│  OS/VS COBOL with the VS COBOL II or COBOL/370 run-time         │
│  libraries.                                                     │
│                                                                 │
│  IMS and DB2 share a common alias name, DSNHLI, for the         │
│  language interface module. You must do the following when you  │
│  concatenate your libraries:                                    │
│   –  If you use IMS with the COBOL option DYNAM, be sure to     │
│      concatenate the IMS library first.                         │
│   –  If you run your application program only under DB2, be     │
│      sure to concatenate the DB2 library first.                 │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

```
┌─ CICS and CAF ────────────────────────────────────────────────┐
│                                                                 │
│  You must specify the option NODYNAM when you compile a COBOL   │
│  program that includes SQL statements. You cannot use DYNAM.    │
│                                                                 │
│  Because stored procedures use CAF, you must also compile       │
│  COBOL stored procedures with the option NODYNAM.               │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

- To avoid truncating numeric values, specify the COBOL compiler option:
  - TRUNC(OPT) if you are certain that the data being moved to each binary variable by the application does not have a larger precision than defined in the PICTURE clause of the binary variable.
  - TRUNC(BIN) if the precision of data being moved to each binary variable might exceed the value in the PICTURE clause.

  DB2 assigns values to COBOL binary integer host variables as if you had specified the COBOL compiler option TRUNC(BIN).
- If a COBOL program contains several entry points or is called several times, the USING clause of the entry statement that executes before the first SQL statement executes must contain the SQLCA and all linkage section entries that any SQL statement uses as host variables.
- The REPLACE statement has no effect on SQL statements. It affects only the COBOL statements that the precompiler generates.
- Do not use COBOL figurative constants (such as ZERO and SPACE), symbolic characters, reference modification, and subscripts within SQL statements.
- Observe the rules in Chapter 2 of *DB2 SQL Reference* when you name SQL identifiers.
- Observe these rules for hyphens:
  - Surround hyphens used as subtraction operators with spaces. DB2 usually interprets a hyphen with no spaces around it as part of a host variable name.
  - You can use hyphens in SQL identifiers under either of the following circumstances:
    - The application program is a local application that runs on DB2 UDB for OS/390 Version 6 or later.

- The application program accesses remote sites, and the local site and remote sites are DB2 UDB for OS/390 Version 6 or later.

- If you include an SQL statement in a COBOL PERFORM ... THRU paragraph and also specify the SQL statement WHENEVER ... GO, then the COBOL compiler returns the warning message IGYOP3094. That message might indicate a problem, depending on the intention behind the code. The usage is not advised.

- If you are using VS COBOL II or COBOL/370 with the option NOCMPR2, then the following additional restrictions apply:

  – All SQL statements and any host variables they reference must be within the first program when using nested programs or batch compilation.

  – DB2 COBOL programs must have a DATA DIVISION and a PROCEDURE DIVISION. Both divisions and the WORKING-STORAGE section must be present in programs that use the DB2 precompiler.

┌─ **Product-sensitive Programming Interface** ──────────────────

If you pass host variables with address changes into a program more than once, then the called program must reset SQL-INIT-FLAG. Resetting this flag indicates that the storage must initialize when the next SQL statement executes. To reset the flag, insert the statement MOVE ZERO TO SQL-INIT-FLAG in the called program's PROCEDURE DIVISION, ahead of any executable SQL statements that use the host variables.

└─ **End of Product-sensitive Programming Interface** ──────────────

Using host variables

You must explicitly declare all host variables used in SQL statements in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program's DATA DIVISION. You must explicitly declare each host variable before its first use in an SQL statement.

You can precede COBOL statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

The names of host variables should be unique within the source data set or member, even if the host variables are in different blocks, classes, or procedures. You can qualify the host variable names with a structure name to make them unique.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

You cannot define host variables, other than indicator variables, as arrays. You can specify OCCURS only when defining an indicator structure. You cannot specify OCCURS for any other type of host variable.

Declaring host variables

Only some of the valid COBOL declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message ″UNDECLARED HOST VARIABLE″.

Numeric host variables: The following figures show the syntax for valid numeric host variable declarations.
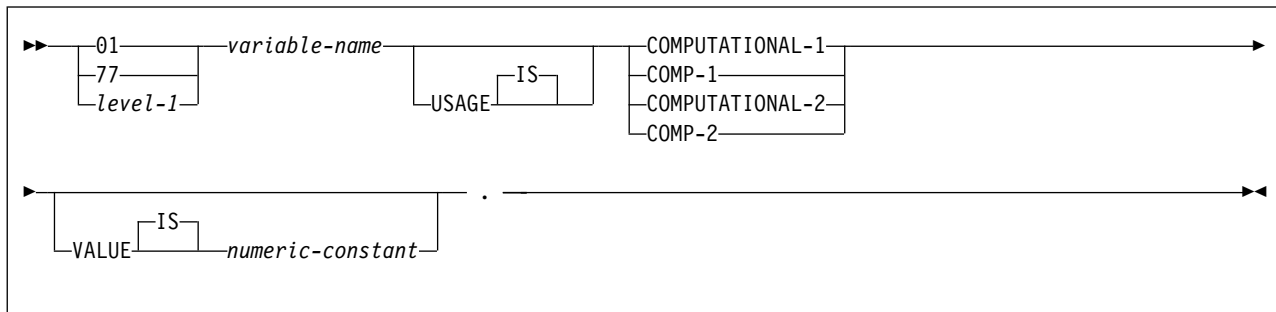
Figure 50. Numeric host variables

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. COMPUTATIONAL-1 and COMP-1 are equivalent.
3. COMPUTATIONAL-2 and COMP-2 are equivalent.
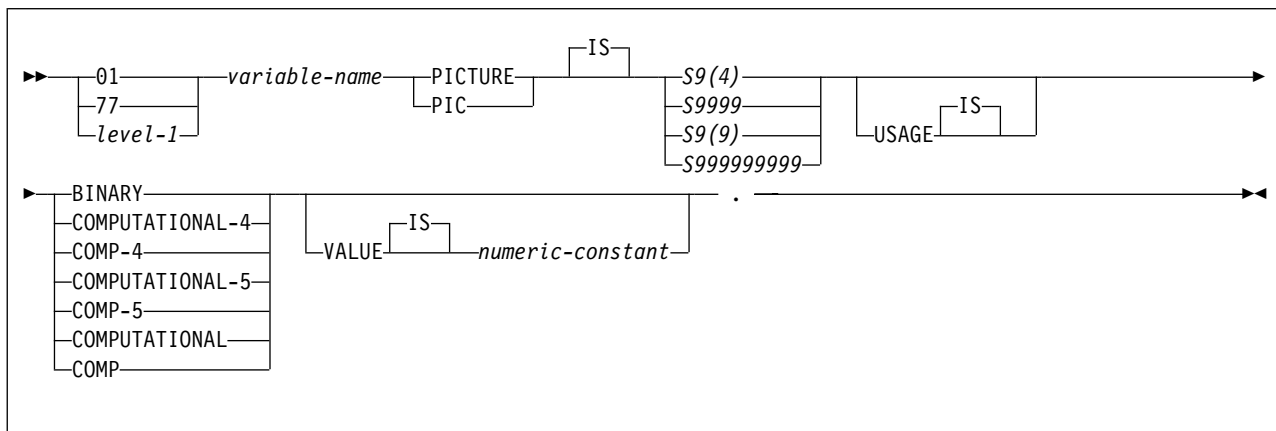
Figure 51. Integer and small integer

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. BINARY, COMP, COMPUTATIONAL, COMPUTATIONAL-4, COMP-4 , COMPUTATIONAL-5, COMP-5 are equivalent.
3. Any specification for scale is ignored.

COBOL
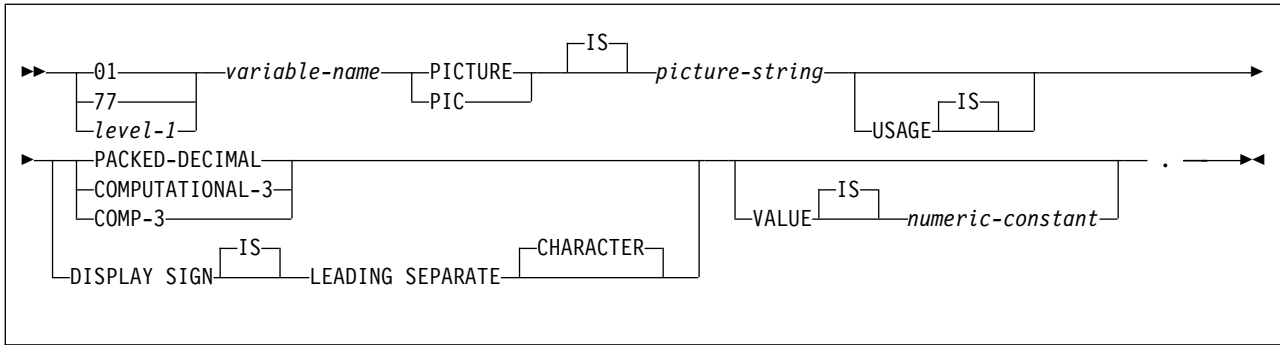
Figure 52. Decimal

> **Notes:**
> 1. *level-1* indicates a COBOL level between 2 and 48.
> 2. PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. The *picture-string* associated with these types must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9) or S9(*i*)V.
> 3. The *picture-string* associated with SIGN LEADING SEPARATE must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9 or S9...9V with *i* instances of 9).

> ***Character host variables:*** There are three valid forms of character host variables:
> *   Fixed-length strings
> *   Varying-length strings
> *   CLOBs

> The following figures show the syntax for forms other than CLOBs. See Figure 59 on page 151 for the syntax of CLOBs.
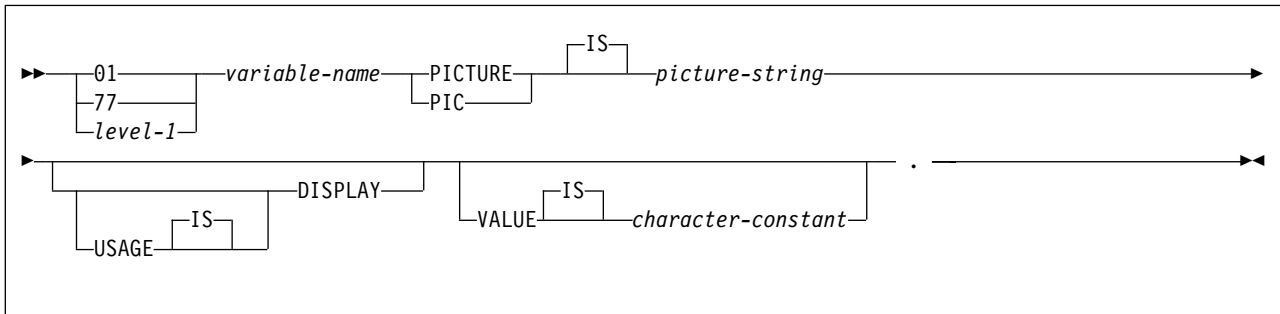
Figure 53. Fixed-length character strings

> **Note:**
>
> *level-1* indicates a COBOL level between 2 and 48.

```
  ►►──┬──01────┬──── variable-name ── . ──────────────────────────►◄
      └─level-1─┘


                            ┌─IS─┐
  ►►──49──var-1──┬─PICTURE─┬──┴────┴──┬─S9(4)─┬──────────────────────┬─BINARY───────────────►
                 └─PIC─────┘          └─S9999─┘  ┌─IS─┐              ├─COMPUTATIONAL-4─┤
                                         └─USAGE─┴────┴─┘            ├─COMP-4──────────┤
                                                                    ├─COMPUTATIONAL-5─┤
                                                                    └─COMP-5──────────┘

  ►──┬───────────────────────────────┬── . ──────────────────────────────►◄
     │      ┌─IS─┐                    │
     └─VALUE─┴────┴──numeric-constant─┘


                            ┌─IS─┐
  ►►──49──var-2──┬─PICTURE─┬──┴────┴── picture-string ─┬────────────────────────────┬──►
                 └─PIC─────┘                           │        ┌─IS─┐              │
                                                       └─USAGE──┴────┴──DISPLAY─────┘

  ►──┬─────────────────────────────────┬── . ──────────────────────────────►◄
     │      ┌─IS─┐                      │
     └─VALUE─┴────┴──character-constant─┘
```
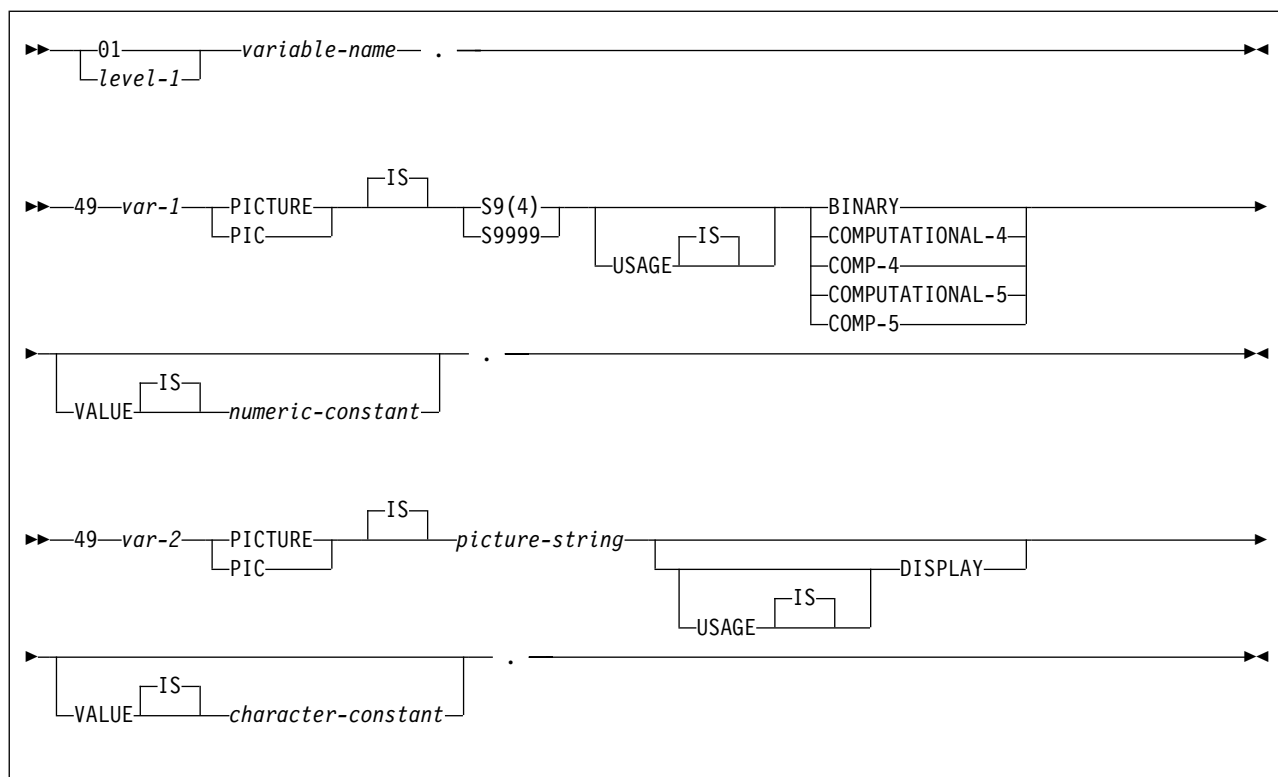
Figure 54. Varying-length character strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.

2. The *picture-string* associated with these forms must be X(*m*) (or XX...X, with *m* instances of X), with 1 <= *m* <= 255 for fixed-length strings; for other strings, *m* cannot be greater than the maximum size of a varying-length character string.

   DB2 uses the full length of the S9(4) variable even though IBM COBOL for MVS and VM only recognizes values up to 9999. This can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(OPT) or NOTRUNC COBOL compiler option (whichever is appropriate) to avoid data truncation.

3. You cannot directly reference *var-1* and *var-2* as host variables.

Graphic character host variables: There are three valid forms for graphic character host variables:
- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following figures show the syntax for forms other than DBCLOBs. See Figure 59 on page 151 for the syntax of DBCLOBs.

COBOL
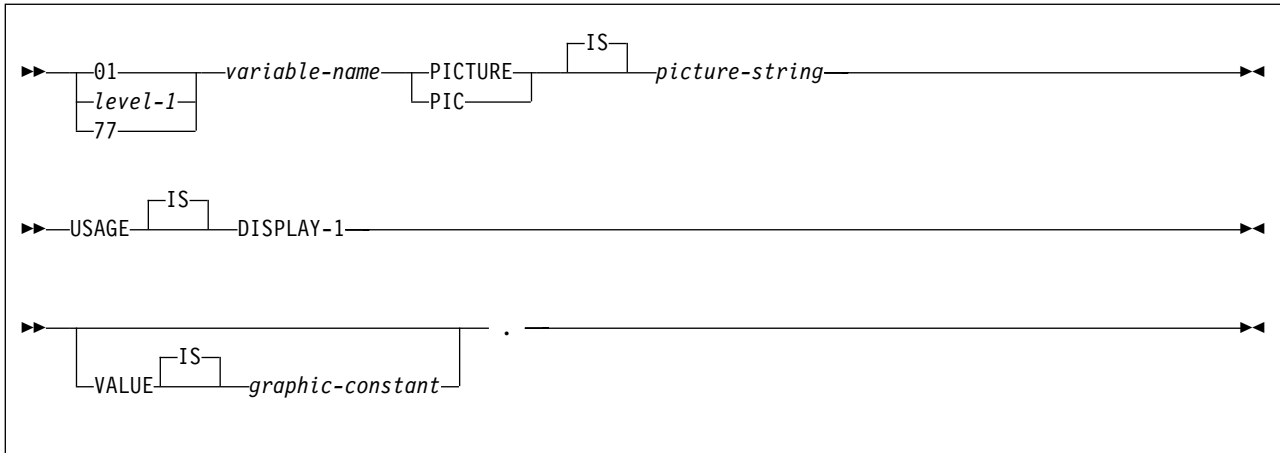
Figure 55. Fixed-length graphic strings

Note:

level-1 indicates a COBOL level between 2 and 48.
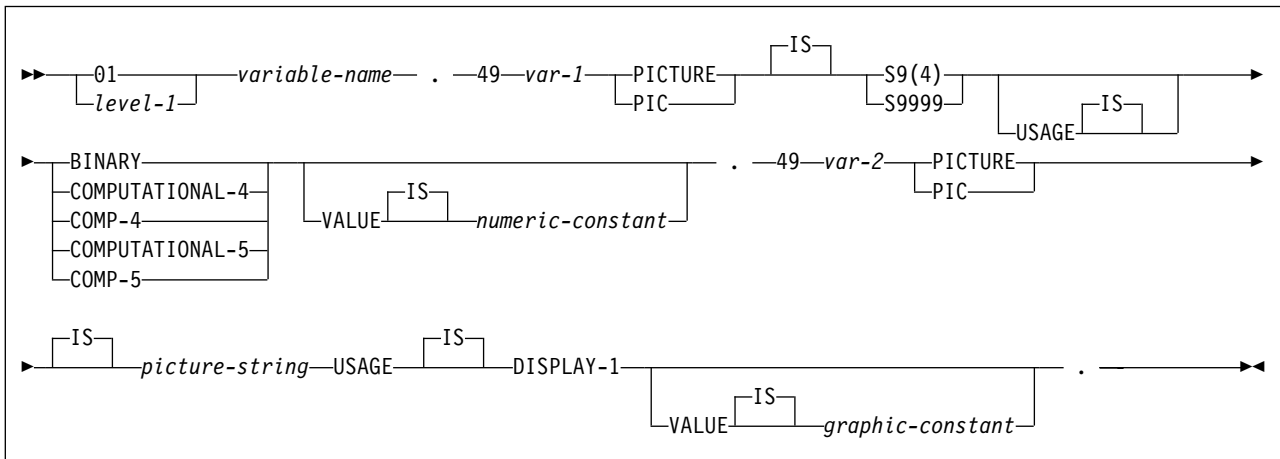
Figure 56. Varying-length graphic strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. The *picture-string* associated with these forms must be G(*m*) (or GG...G, with *m* instances of G), with 1 <= *m* <= 127 for fixed-length strings. You can use *N* in place of G for COBOL graphic variable declarations. If you use N for graphic variable declarations, USAGE DISPLAY-1 is optional. For strings other than fixed-length, *m* cannot be greater than the maximum size of a varying-length graphic string.

   DB2 uses the full size of the S9(4) variable even some COBOL implementations restrict the maximum length of varying-length graphic string to 9999. This can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length graphic strings to 9999. Consider using the TRUNC(OPT) or NOTRUNC COBOL compiler option (which ever is appropriate) to avoid data truncation.
3. You cannot directly reference *var*-1 and *var*-2 as host variables.

Result set locators: The following figure shows the syntax for declarations of result set locators. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for a discussion of how to use these host variables.

```
►►─01─variable-name─────────────────────SQL TYPE IS RESULT-SET-LOCATOR VARYING─.───────►◄
              └USAGE─┐
                     └IS┘
```
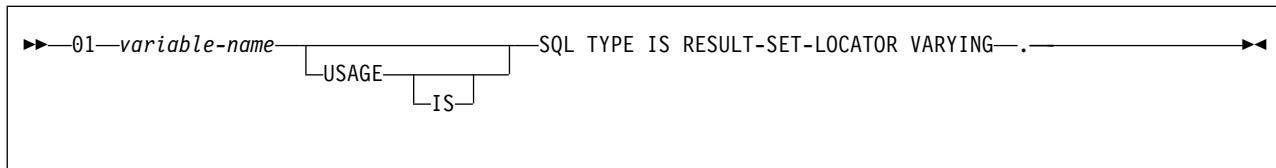
Figure 57. Result set locators

Table Locators: The following figure shows the syntax for declarations of table locators. See "Accessing transition tables in a user-defined function or stored procedure" on page 278 for a discussion of how to use these host variables.

```
►►──01──────────variable-name────────────SQL TYPE IS─TABLE LIKE─table-name─────────────►
    └level-1─┘              └USAGE─┐
                                   └IS┘

►─AS LOCATOR──.──────────────────────────────────────────────────────────────────────►◄
```
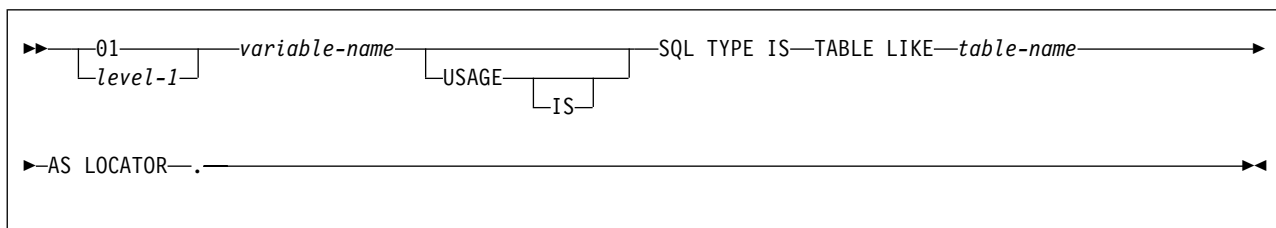
Figure 58. Table locators

Note:

level-1 indicates a COBOL level between 2 and 48.

LOB Variables and Locators: The following figure shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►──01──────────variable-name───────────SQL─TYPE─IS──────────────────────────────────►
    └level-1─┘              └USAGE─┐
                                   └IS┘

►──┬─BINARY LARGE OBJECT─┬───(─length─┬───┬─)───────────────────────────────────────►◄
   │ └BLOB──────────────┘            ├─K─┤
   │ ┌CHARACTER LARGE OBJECT┐        ├─M─┤
   │ ├CHAR LARGE OBJECT──────┤       └─G─┘
   │ └CLOB──────────────────┘
   ├─DBCLOB───────────────────┘
   ├─BLOB-LOCATOR───────
   ├─CLOB-LOCATOR───────
   └─DBCLOB-LOCATOR─────
```

Figure 59. LOB variables and locators

Note:

level-1 indicates a COBOL level between 2 and 48.

ROWIDs: The following figure shows the syntax for declarations of ROWID variables. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►──┬─01──────┬──variable-name──┬──────────────────┬──SQL TYPE IS──ROWID──.──────────────►◄
    └─level-1─┘                 └─USAGE──┬──────┬──┘
                                         └─IS──┘
```

Figure 60. ROWID variables

Note:

level-1 indicates a COBOL level between 2 and 48.

Using host structures

A COBOL host structure is a named set of host variables defined in your program's WORKING-STORAGE SECTION or LINKAGE SECTION. COBOL host structures have a maximum of two levels, even though the host structure might occur within a structure with multiple levels. However, you can declare a varying-length character string, which must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. In the following example, B is the name of a host structure consisting of the elementary items C1 and C2.

```
01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
```

When you write an SQL statement using a qualified host variable name (perhaps to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B.

The precompiler does not recognize host variables or host structures on any subordinate levels after one of these items:
• A COBOL item that must begin in area A
• Any SQL statement (except SQL INCLUDE)
• Any SQL statement within an included member

When the precompiler encounters one of the above items in a host structure, it therefore considers the structure to be complete.

Figure 61 on page 153 shows the syntax for valid host structures.
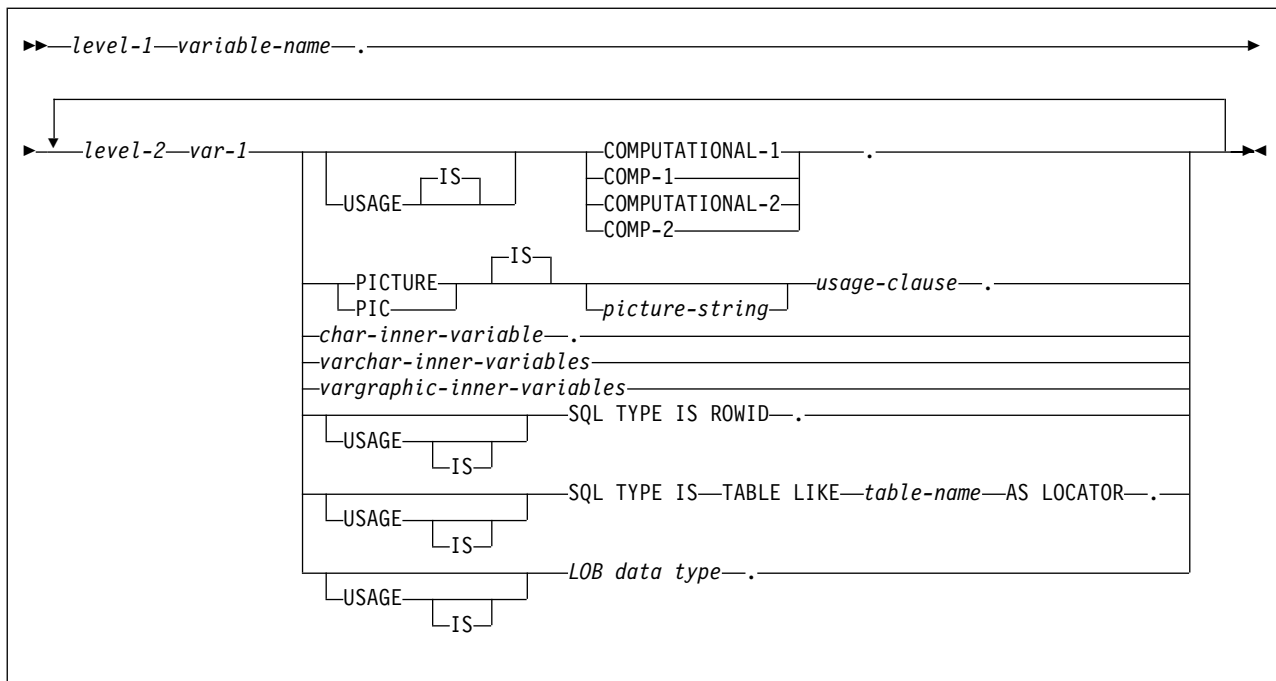
Figure 61. Host structures in COBOL
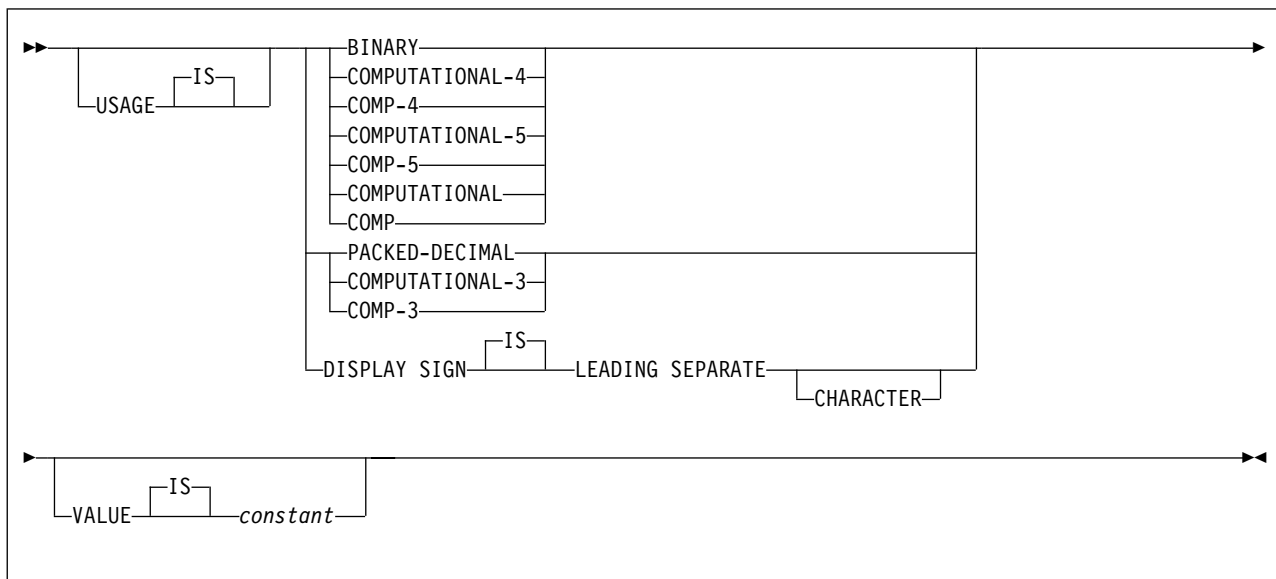
Figure 62. Usage-clause

COBOL
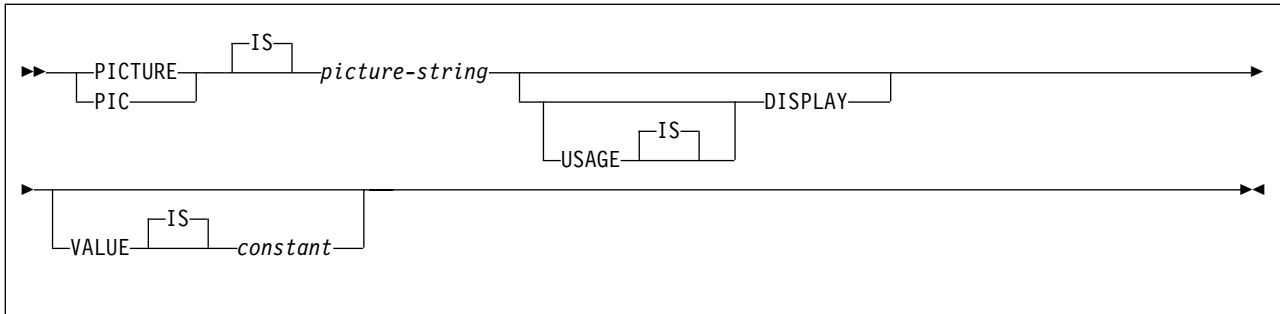
Figure 63. CHAR-inner-variable
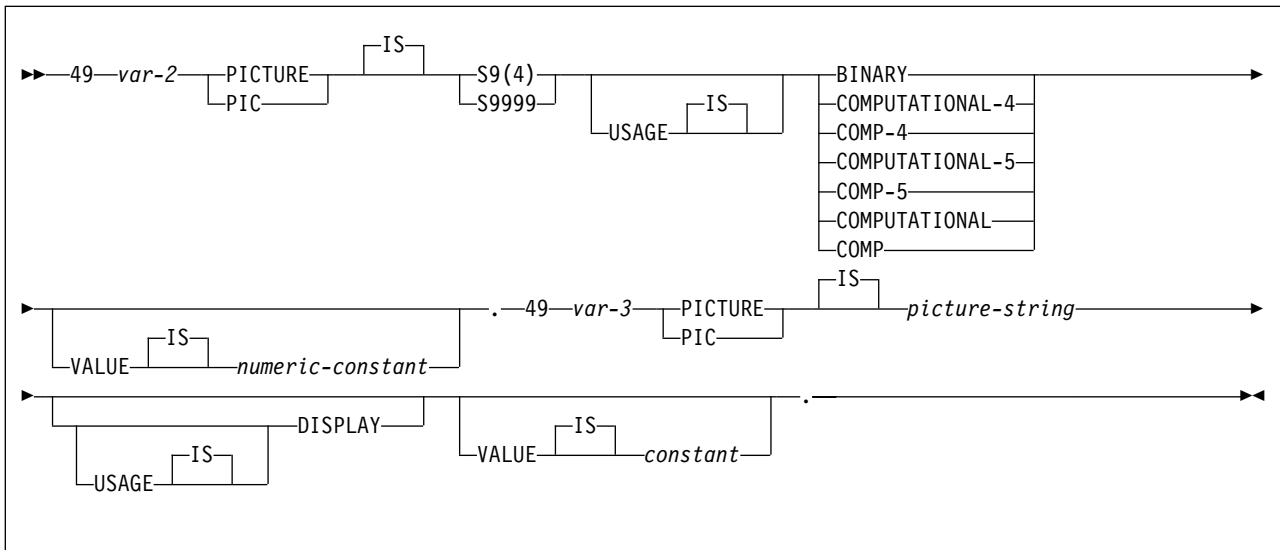
Figure 64. VARCHAR-inner-variables
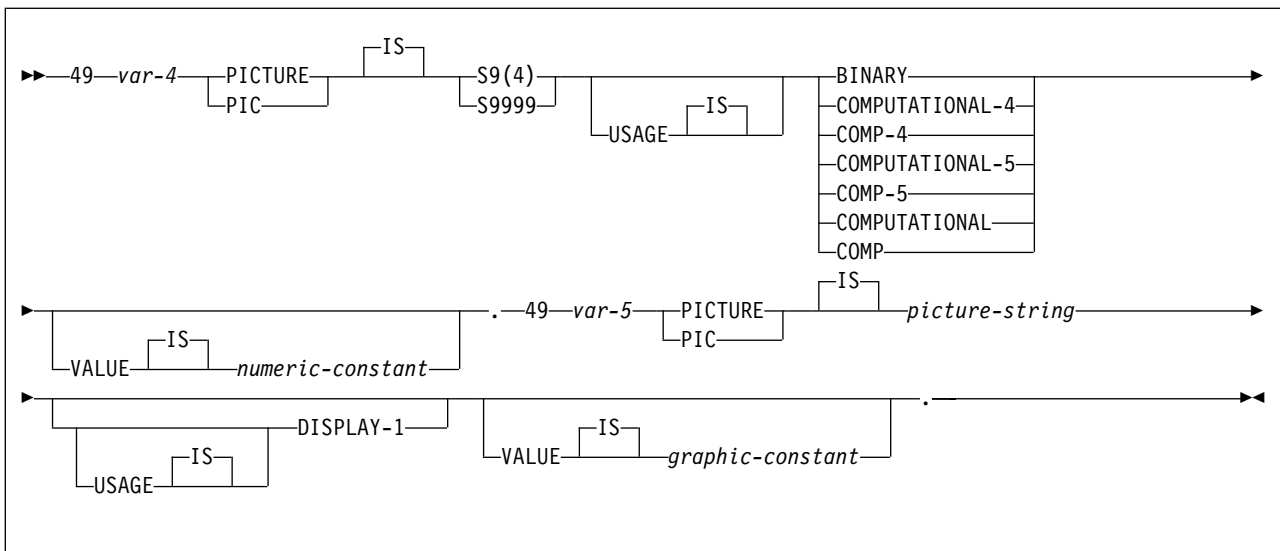
Figure 65. VARGRAPHIC-inner-variables

```
►►─SQL─TYPE─IS─┬────────────┬─BINARY LARGE OBJECT──────┬───(─length─┬───────┬─)─┬───────►◄
              │            ├─BLOB─────────────────────┤            ├─K─┤
              │            ├─CHARACTER LARGE OBJECT────┤            ├─M─┤
              │            ├─CHAR LARGE OBJECT─────────┤            └─G─┘
              │            ├─CLOB──────────────────────┤
              │            └─DBCLOB────────────────────┘
              ├─BLOB-LOCATOR──────┤
              ├─CLOB-LOCATOR──────┤
              └─DBCLOB-LOCATOR────┘
```
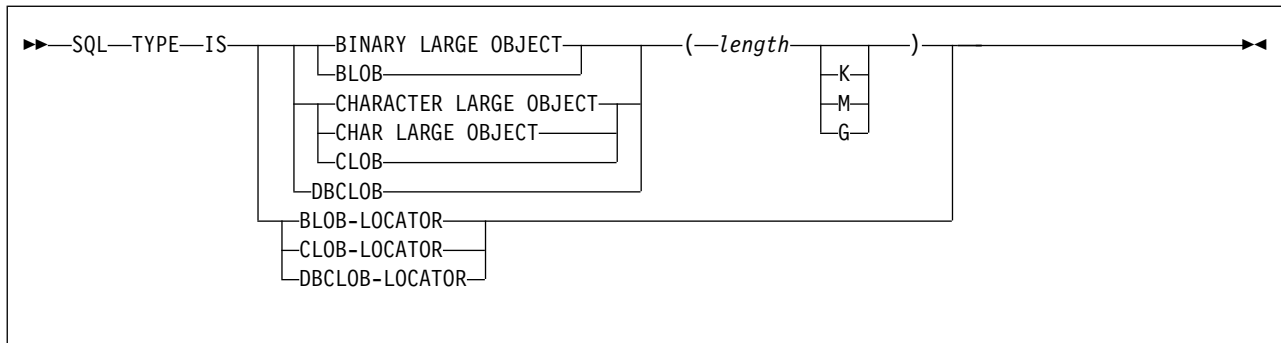
Figure 66. LOB data type

Notes:

1. *level-1* indicates a COBOL level between 1 and 47.
2. *level-2* indicates a COBOL level between 2 and 48.
3. For elements within a structure use any level 02 through 48 (rather than 01 or 77), up to a maximum of two levels.
4. Using a FILLER or optional FILLER item within a host structure declaration can invalidate the whole structure.
5. You cannot use *picture-string* for floating point elements but must use it for other data types.

Determining equivalent SQL and COBOL data types

Table 12 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 12. Sql data types the precompiler uses for COBOL declarations

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
COMP-1	480	4	REAL or FLOAT(n) $1<=n<=21$
COMP-2	480	8	DOUBLE PRECISION, or FLOAT(n) $22<=n<=53$
S9(i)V9(d) COMP-3 or S9(i)V9(d) PACKED-DECIMAL	484	$i+d$ in byte 1, d in byte 2	DECIMAL($i+d,d$) or NUMERIC($i+d,d$)
S9(i)V9(d) DISPLAY SIGN LEADING SEPARATE	504	$i+d$ in byte 1, d in byte 2	No exact equivalent. Use DECIMAL($i+d,d$) or NUMERIC($i+d,d$)
S9(4) COMP-4, S9(4) COMP-5, or BINARY	500	2	SMALLINT
S9(9) COMP-4, S9(9) COMP-5, or BINARY	496	4	INTEGER
Fixed-length character data	452	m	CHAR(m)
Varying-length character data $1<=m<=255$	448	m	VARCHAR(m)
Varying-length character data $m>255$	456	m	VARCHAR(m)

COBOL

Table 12. Sql data types the precompiler uses for COBOL declarations (continued)

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
Fixed-length graphic data	468	*m*	GRAPHIC(*m*)
Varying-length graphic data 1<=*m*<=127	464	*m*	VARGRAPHIC(*m*)
Varying-length graphic data *m*>127	472	*m*	VARGRAPHIC(*m*)
SQL TYPE IS RESULT-SET-LOCATOR	972	4	Result set locator[1]
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table locator[1]
SQL TYPE IS BLOB-LOCATOR	960	4	BLOB locator[1]
SQL TYPE IS CLOB-LOCATOR	964	4	CLOB locator[1]
USAGE IS SQL TYPE IS DBCLOB-LOCATOR	968	4	DBCLOB locator[1]
USAGE IS SQL TYPE IS BLOB(*n*) 1≤*n*≤2147483647	404	*n*	BLOB(*n*)
USAGE IS SQL TYPE IS CLOB(*n*) 1≤*n*≤2147483647	408	*n*	CLOB(*n*)
USAGE IS SQL TYPE IS DBCLOB(*m*) 1≤*m*≤1073741823[2]	412	*n*	DBCLOB(*m*)[2]
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. Do not use this data type as a column type.

2. *m* is the number of double-byte characters.

Table 13 helps you define host variables that receive output from the database. You can use the table to determine the COBOL data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 13 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 13. SQL data types mapped to typical COBOL declarations

SQL Data Type	COBOL Data Type	Notes
SMALLINT	S9(4) COMP-4, S9(4) COMP-5, or BINARY	
INTEGER	S9(9) COMP-4, S9(9) COMP-5, or BINARY	

Table 13. SQL data types mapped to typical COBOL declarations (continued)

SQL Data Type	COBOL Data Type	Notes

DECIMAL(*p,s*) or NUMERIC(*p,s*)	S9(*p-s*)V9(*s*) COMP-3 or S9(*p-s*)V9(*s*)
	  PACKED-DECIMAL
	  DISPLAY SIGN
	  LEADING SEPARATE
		p is precision; *s* is scale. 0<=*s*<=*p*<=31. If *s*=0, use S9(*p*)V or S9(*p*). If s=p, use SV9(*s*). If the COBOL compiler does not support 31–digit decimal numbers, there is no exact equivalent. Use COMP-2.

REAL or FLOAT (*n*)	COMP-1	1<=*n*<=21
DOUBLE PRECISION, DOUBLE or FLOAT (*n*)	COMP-2	22<=*n*<=53
CHAR(*n*)	Fixed-length character string. For example, `01 VAR-NAME PIC X(n).`	1<=*n*<=255

VARCHAR(*n*)	Varying-length character string. For example,
	`01 VAR-NAME.`
	`  49 VAR-LEN PIC S9(4) USAGE BINARY.`
	`  49 VAR-TEXT PIC X(n).`
		The inner variables must have a level of 49.

GRAPHIC(*n*)	Fixed-length graphic string. For example,
	`01 VAR-NAME PIC G(n)`
	`   USAGE IS DISPLAY-1.`
		n refers to the number of double-byte characters, not to the number of bytes. 1<=*n*<=127

VARGRAPHIC(*n*)	Varying-length graphic string. For example,
	`01 VAR-NAME.`
	`  49 VAR-LEN PIC S9(4) USAGE BINARY.`
	`  49 VAR-TEXT PIC G(n)`
	`     USAGE IS DISPLAY-1.`
		n refers to the number of double-byte characters, not to the number of bytes.
		
		The inner variables must have a level of 49.

DATE	Fixed-length character string of length *n*. For example, `01 VAR-NAME PIC X(n).`	If you are using a date exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 10.
TIME	Fixed-length character string of length *n*. For example, `01 VAR-NAME PIC X(n).`	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	Fixed-length character string of length of length *n*. For example, `01 VAR-NAME PIC X(n).`	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT-SET-LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	USAGE IS SQL TYPE IS BLOB-LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	USAGE IS SQL TYPE IS CLOB-LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.

COBOL

Table 13. SQL data types mapped to typical COBOL declarations (continued)

SQL Data Type	COBOL Data Type	Notes
DBCLOB locator	USAGE IS SQL TYPE IS DBCLOB-LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB(*n*)	USAGE IS SQL TYPE IS BLOB(*n*)	1≤n≤2147483647
CLOB(*n*)	USAGE IS SQL TYPE IS CLOB(*n*)	1≤n≤2147483647
DBCLOB(*n*)	USAGE IS SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. 1≤n≤1073741823
ROWID	SQL TYPE IS ROWID	

Notes on COBOL variable declaration and usage

You should be aware of the following when you declare COBOL variables.

SQL data types with no COBOL equivalent: If you are using a COBOL compiler that does not support decimal numbers of more than 18 digits, use one of the following data types to hold values of greater than 18 digits:

- A decimal variable with a precision less than or equal to 18, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, then the fractional part of the value could be truncated.

- An integer or a floating-point variable, which converts the value. If you choose integer, you lose the fractional part of the number. If the decimal number could exceed the maximum value for an integer or, if you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. Hence, when you assign a decimal number to a floating point variable, the result could be different from the original number.

- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special Purpose COBOL Data Types: The locator data types are COBOL data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator
　　　　　"Chapter 24. Using stored procedures for client/server processing" on page 525

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 278

LOB locators "Chapter 13. Programming for large objects (LOBs)" on page 229

Level 77 data description entries: One or more REDEFINES entries can follow any level 77 data description entry. However, you cannot use the names in these entries in SQL statements. Entries with the name FILLER are ignored.

SMALLINT and INTEGER data types: In COBOL, you declare the SMALLINT and INTEGER data types as a number of decimal digits. DB2 uses the full size of the integers (in a way that is similar to processing with the COBOL options TRUNC(OPT) or NOTRUNC) and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration.

However, this can cause data truncation when COBOL statements execute. Ensure that the size of numbers in your application is within the declared number of digits.

For small integers that can exceed 9999, use S9(5) COMP. For large integers that can exceed 999,999,999, use S9(10) COMP-3 to obtain the decimal data type. If you use COBOL for integers that exceed the COBOL PICTURE, then specify the column as decimal to ensure that the data types match and perform well.

Overflow: Be careful of overflow. For example, suppose you retrieve an INTEGER column value into a PICTURE S9(4) host variable and the column value is larger than 32767 or smaller than -32768. You get an overflow warning or an error, depending on whether you specify an indicator variable.

VARCHAR and VARGRAPHIC data types: If your varying-length character host variables receive values whose length is greater than 9999 characters, compile the applications in which you use those host variables with the option TRUNC(BIN). TRUNC(BIN) lets the length field for the character string receive a value of up to 32767.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a PICTURE X(70) host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double precision floating-point or decimal column value into a PIC S9(8) COMP host variable removes any fractional part of the value.

Similarly, retrieving a DB2 DECIMAL number into a COBOL equivalent variable could truncate the value. This happens because a DB2 DECIMAL value can have up to 31 digits, but a COBOL decimal number can have up to only 18 digits.

Determining compatibility of SQL and COBOL data types

COBOL host variables used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, REAL, or DOUBLE PRECISION column is compatible with a COBOL host variable of PICTURE S9(4), PICTURE S9(9), COMP-3, COMP-1, COMP-4, COMP-5, COMP-2, BINARY, or PACKED-DECIMAL. A DECIMAL column is also compatible with a COBOL host variable declared as DISPLAY SIGN IS LEADING SEPARATE.
- Character data types are compatible with each other: A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length COBOL character host variable.
- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.

- Graphic data types are compatible with each other: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying length COBOL graphic string host variable.

- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:

  - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column

  - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.

  - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.

  - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.

  - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.

- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying length COBOL character host variable.

- A BLOB column or a BLOB locator is compatible only with a BLOB host variable.

- The ROWID column is compatible only with a ROWID host variable.

- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. For information on assigning and comparing distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables

An indicator variable is a 2-byte integer (PIC S9(4) USAGE BINARY). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. You can define indicator variables as scalar variables or as array elements in a structure form or as an array variable using a single level OCCURS clause. For more information about indicator variables, see "Using indicator variables with host variables" on page 70.

Example

Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLS-CD,
                               :DAY :DAY-IND,
                               :BGN :BGN-IND,
                               :END :END-IND
END-EXEC.
```

You can declare the variables as follows:

```
77 CLS-CD   PIC X(7).
77 DAY      PIC S9(4) BINARY.
77 BGN      PIC X(8).
77 END      PIC X(8).
77 DAY-IND  PIC S9(4) BINARY.
77 BGN-IND  PIC S9(4) BINARY.
77 END-IND  PIC S9(4) BINARY.
```

The following figure shows the syntax for a valid indicator variable.
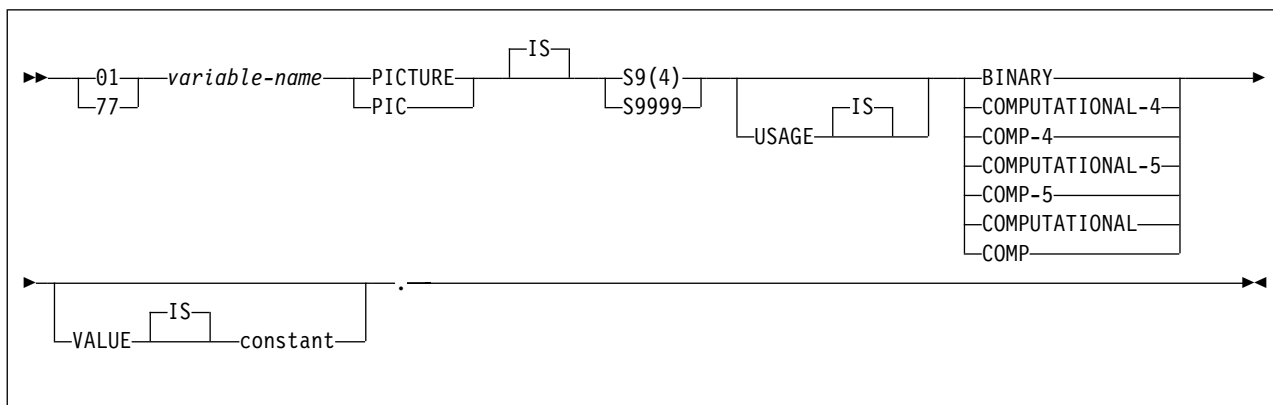
Figure 67. Indicator variable

The following figure shows the syntax for valid indicator array declarations.
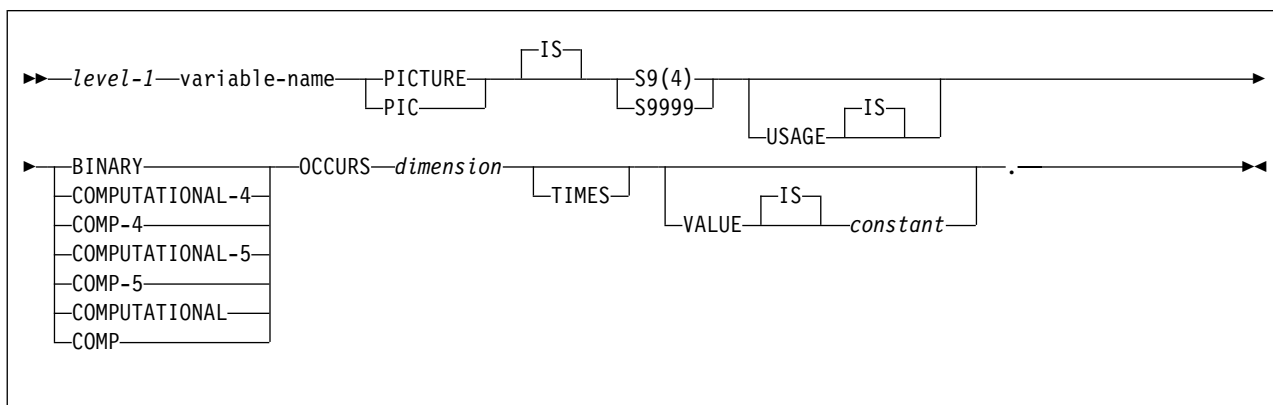
Figure 68. Host structure indicator array

> **Note:** *level-1* must be an integer between 2 and 48.

Handling SQL error return codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL error return codes" on page 76.

```
┌─ DSNTIAR syntax ─────────────────────────────────────────────────┐
│  CALL 'DSNTIAR' USING sqlca message lrecl.                        │
└──────────────────────────────────────────────────────────────────┘
```

The DSNTIAR parameters have the following meanings:

sqlca
> An SQL communication area.

message
> An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

> The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
01  ERROR-MESSAGE.
      02  ERROR-LEN    PIC S9(4)  COMP VALUE +1320.
      02  ERROR-TEXT   PIC X(132) OCCURS 10 TIMES
                                  INDEXED BY ERROR-INDEX.
77  ERROR-TEXT-LEN     PIC S9(9)  COMP VALUE +132.
 ⋮
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

> where ERROR-MESSAGE is the name of the message output area containing 10 lines of length 132 each, and ERROR-TEXT-LEN is the length of each line.

lrecl
> A fullword containing the logical record length of output messages, between 72 and 240.

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSN8BC3, contained in the library DSN8710.SDSNSAMP. See "Appendix B. Sample applications" on page 829 for instructions on how to access and print the source code for the sample program.

CICS

If you call DSNTIAR dynamically from a CICS VS COBOL II or CICS
COBOL/370 application program, be sure you do the following:
- Compile the COBOL application with the NODYNAM option.
- Define DSNTIAR in the CSD.

If your CICS application requires CICS storage handling, you must use the
subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL 'DSNTIAC' USING eib commarea sqlca msg lrecl.
```

DSNTIAC has extra parameters, which you must use for calls to routines that
use CICS commands.

eib   EXEC interface block

commarea   communication area

For more information on these new parameters, see the appropriate
application programming guide for CICS. The remaining parameter
descriptions are the same as those for DSNTIAR. Both DSNTIAC and
DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you
must also define them in the CSD. For an example of CSD entry generation
statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which
assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Considerations for object-oriented extensions in COBOL

When you use object-oriented extensions in an IBM COBOL for MVS & VM
application, be aware of the following:

Where to Place SQL Statements in Your Application: An IBM COBOL for MVS &
VM source data set or member can contain the following elements:
- Multiple programs
- Multiple class definitions, each of which contains multiple methods

You can put SQL statements in only the first program or class in the source data
set or member. However, you can put SQL statements in multiple methods within a
class. If an application consists of multiple data sets or members, each of the data
sets or members can contain SQL statements.

Where to Place the SQLCA, SQLDA, and Host Variable Declarations: You can
put the SQLCA, SQLDA, and SQL host variable declarations in the
WORKING-STORAGE SECTION of a program, class, or method. An SQLCA or
SQLDA in a class WORKING-STORAGE SECTION is global for all the methods of
the class. An SQLCA or SQLDA in a method WORKING-STORAGE SECTION is
local to that method only.

If a class and a method within the class both contain an SQLCA or SQLDA, the
method uses the SQLCA or SQLDA that is local.

COBOL

> ***Rules for Host Variables:*** You can declare COBOL variables that are used as host variables in the WORKING-STORAGE SECTION or LINKAGE-SECTION of a program, class, or method. You can also declare host variables in the LOCAL-STORAGE SECTION of a method. The scope of a host variable is the method, class, or program within which it is defined.

Coding SQL statements in a FORTRAN application

This section helps you with the programming techniques that are unique to coding SQL statements within a FORTRAN program.

Defining the SQL communication area

A FORTRAN program that contains SQL statements must include one or both of the following host variables:

- An SQLCOD variable declared as INTEGER*4
- An SQLSTA (or SQLSTATE) variable declared as CHARACTER*5

Or,

- An SQLCA, which contains the SQLCOD and SQLSTA variables.

DB2 sets the SQLCOD and SQLSTA (or SQLSTATE) values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCOD and SQLSTA (or SQLSTATE) variables.

Whether you define SQLCOD or SQLSTA (or SQLSTATE), or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)
When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTA (or SQLSTATE) variable, it must not be an element of a structure. You must declare the host variables SQLCOD and SQLSTA (or SQLSTATE) within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If you specify STDSQL(NO)
When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a FORTRAN program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:
- CALL...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*

- DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
- DESCRIBE INPUT *statement-name* INTO *descriptor-name*
- DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
- DESCRIBE TABLE *host-variable* INTO *descriptor-name*
- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- PREPARE...INTO *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. DB2 does not support the INCLUDE SQLDA statement for FORTRAN programs. If present, an error message results.

You can have a FORTRAN program call a subroutine (written in C, PL/I or assembler language) that uses the DB2 INCLUDE SQLDA statement to define the SQLDA and also includes the necessary SQL statements for the dynamic SQL functions you wish to perform. See "Chapter 23. Coding dynamic SQL in application programs" on page 495 for more information about dynamic SQL.

You must place SQLDA declarations before the first SQL statement that references the data descriptor.

Embedding SQL statements

FORTRAN source statements must be fixed-length 80-byte records. The DB2 precompiler does not support free-form source input.

You can code SQL statements in a FORTRAN program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

Each SQL statement in a FORTRAN program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code the statement UPDATE in a FORTRAN program as follows:

```
 EXEC SQL
C  UPDATE DSN8710.DEPT
C  SET MGRNO = :MGRNUM
C  WHERE DEPTNO = :INTDEPT
```

You cannot follow an SQL statement with another SQL statement or FORTRAN statement on the same line.

FORTRAN does not require blanks to delimit words within a statement, but the SQL language requires blanks. The rules for embedded SQL follow the rules for SQL syntax, which require you to use one or more blanks as a delimiter.

Comments: You can include FORTRAN comment lines within embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

The DB2 precompiler does not support the exclamation point (!) as a comment recognition character in FORTRAN programs.

FORTRAN

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for FORTRAN statements, except that you must specify EXEC SQL on one line. The SQL examples in this section have Cs in the sixth column to indicate that they are continuations of the statement EXEC SQL.

Declaring tables and views: Your FORTRAN program should also include the statement DECLARE TABLE to describe each table and view the program accesses.

Dynamic SQL in a FORTRAN program: In general, FORTRAN programs can easily handle dynamic SQL statements. SELECT statements can be handled if the data types and the number of fields returned are fixed. If you want to use variable-list SELECT statements, you need to use an SQLDA. See "Defining SQL descriptor areas" on page 164 for more information on SQLDA.

You can use a FORTRAN character variable in the statements PREPARE and EXECUTE IMMEDIATE, even if it is fixed-length.

Including code: To include SQL statements or FORTRAN host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements. You cannot use the FORTRAN INCLUDE compiler directive to include SQL statements or FORTRAN host variable declarations.

Margins: Code the SQL statements between columns 7 through 72, inclusive. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid FORTRAN name for a host variable. Do not use external entry names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Do not use the word DEBUG, except when defining a FORTRAN DEBUG packet. Do not use the words FUNCTION, IMPLICIT, PROGRAM, and SUBROUTINE to define variables.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement labels: You can specify statement numbers for SQL statements in columns 1 to 5. However, during program preparation, a labelled SQL statement generates a FORTRAN statement CONTINUE with that label before it generates the code that executes the SQL statement. Therefore, a labelled SQL statement should never be the last statement in a DO loop. In addition, you should not label SQL statements (such as INCLUDE and BEGIN DECLARE SECTION) that occur before the first executable SQL statement because an error might occur.

WHENEVER statement: The target for the GOTO clause in the SQL statement WHENEVER must be a label in the FORTRAN source and must refer to a statement in the same subprogram. The statement WHENEVER only applies to SQL statements in the same subprogram.

Special FORTRAN considerations: The following considerations apply to programs written in FORTRAN:

- You cannot use the @PROCESS statement in your source code. Instead, specify the compiler options in the PARM field.
- You cannot use the SQL INCLUDE statement to include the following statements: PROGRAM, SUBROUTINE, BLOCK, FUNCTION, or IMPLICIT.

DB2 supports Version 3 Release 1 of VS FORTRAN with the following restrictions:

- There is no support for the parallel option. Applications that contain SQL statements must not use FORTRAN parallelism.
- You cannot use the byte data type within embedded SQL, because byte is not a recognizable host data type.

Using host variables

You must explicitly declare all host variables used in SQL statements. You cannot implicitly declare any host variables through default typing or by using the IMPLICIT statement. You must explicitly declare each host variable before its first use in an SQL statement.

You can precede FORTRAN statements that define the host variables with a BEGIN DECLARE SECTION statement and follow the statements with an END DECLARE SECTION statement. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

The names of host variables should be unique within the program, even if the host variables are in different blocks, functions, or subroutines.

When you declare a character host variable, you must not use an expression to define the length of the character variable. You can use a character host variable with an undefined length (for example, CHARACTER *(*)). The length of any such variable is determined when its associated SQL statement executes.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Host variables must be scalar variables; they cannot be elements of vectors or arrays (subscripted variables).

You must be careful when calling subroutines that might change the attributes of a host variable. Such alteration can cause an error while the program is running. See Appendix C of *DB2 SQL Reference* for more information.

Declaring host variables

Only some of the valid FORTRAN declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message ″UNDECLARED HOST VARIABLE″.

Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations.

FORTRAN
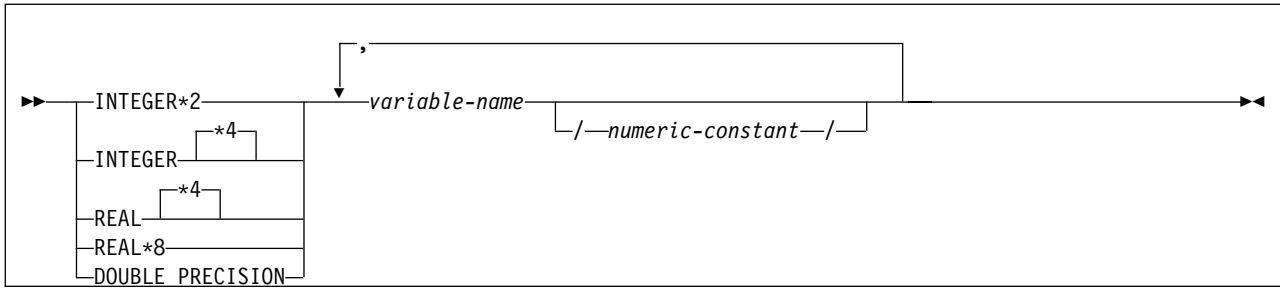
Figure 69. Numeric host variables

Character host variables: The following figure shows the syntax for valid character host variable declarations other than CLOBs. See Figure 72 for the syntax of CLOBs.
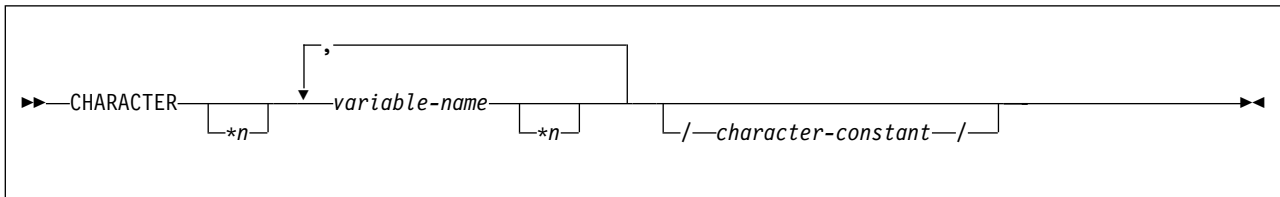
Figure 70. Character host variables

Result set locators: The following figure shows the syntax for declarations of result set locators. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for a discussion of how to use these host variables.
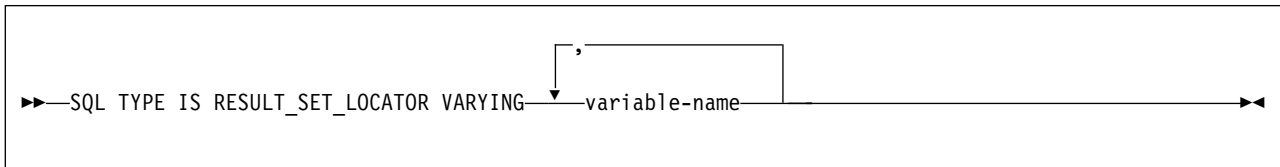
Figure 71. Result set locators

LOB Variables and Locators: The following figure shows the syntax for declarations of BLOB and CLOB host variables and locators. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.
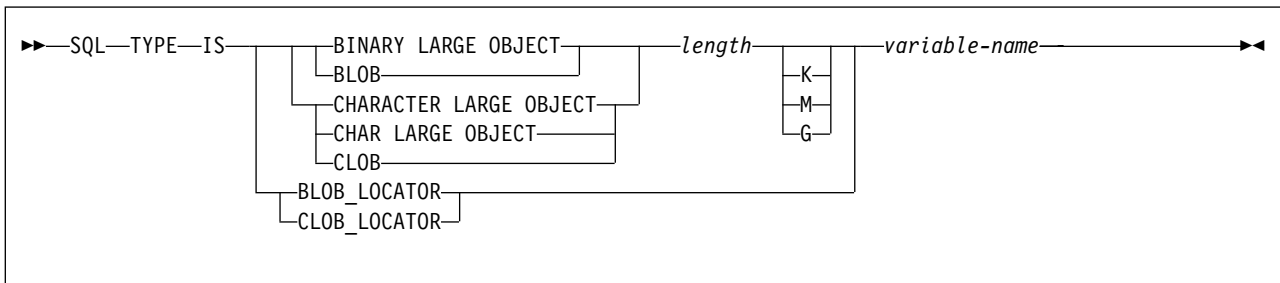
Figure 72. LOB variables and locators

ROWIDs: The following figure shows the syntax for declarations of ROWID variables. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►──SQL TYPE IS──ROWID──variable-name─────────────────────────────────────────►◄
```
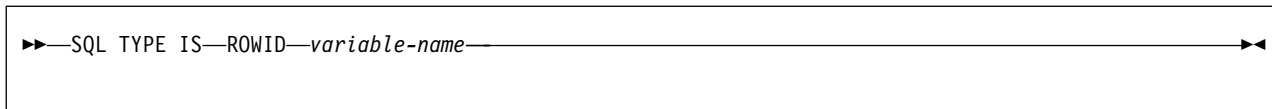
Figure 73. ROWID variables

Determining equivalent SQL and FORTRAN data types

Table 14 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 14. SQL data types the precompiler uses for FORTRAN declarations
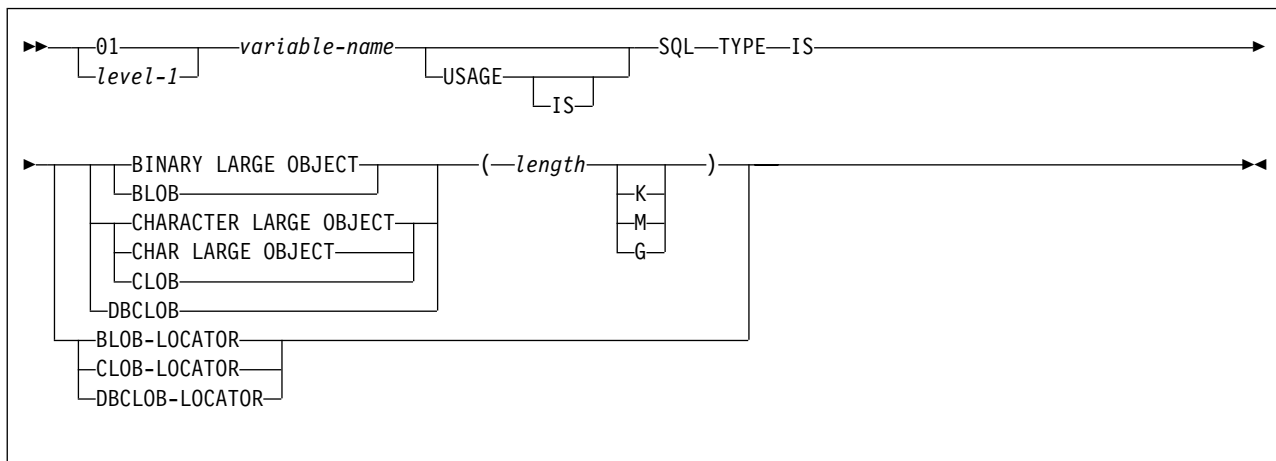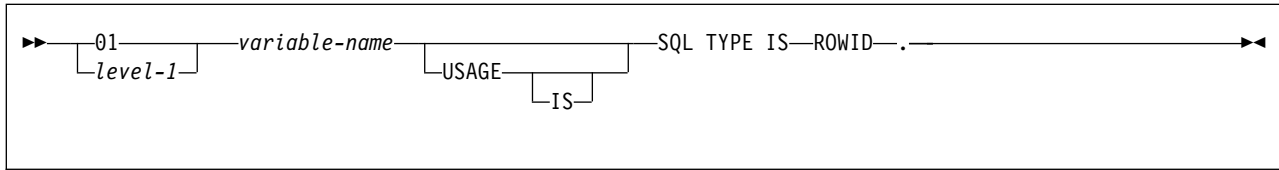
FORTRAN Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
INTEGER*2	500	2	SMALLINT
INTEGER*4	496	4	INTEGER
REAL*4	480	4	FLOAT (single precision)
REAL*8	480	8	FLOAT (double precision)
CHARACTER*n	452	n	CHAR(n)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator. Do not use this data type as a column type.
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator. Do not use this data type as a column type.
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator. Do not use this data type as a column type.
SQL TYPE IS BLOB(n) 1≤n≤2147483647	404	n	BLOB(n)
SQL TYPE IS CLOB(n) 1≤n≤2147483647	408	n	CLOB(n)
SQL TYPE IS ROWID	904	40	ROWID

Table 15 on page 170 helps you define host variables that receive output from the database. You can use the table to determine the FORTRAN data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 15 on page 170 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

FORTRAN

Table 15. SQL data types mapped to typical FORTRAN declarations

SQL Data Type	FORTRAN Equivalent	Notes
SMALLINT	INTEGER*2	
INTEGER	INTEGER*4	
DECIMAL(*p,s*) or NUMERIC(*p,s*)	no exact equivalent	Use REAL*8
FLOAT(*n*) single precision	REAL*4	1<=*n*<=21
FLOAT(*n*) double precision	REAL*8	22<=*n*<=53
CHAR(*n*)	CHARACTER**n*	1<=*n*<=255
VARCHAR(*n*)	no exact equivalent	Use a character host variable large enough to contain the largest expected VARCHAR value.
GRAPHIC(*n*)	not supported	
VARGRAPHIC(*n*)	not supported	
DATE	CHARACTER**n*	If you are using a date exit routine, *n* is determined by that routine; otherwise, *n* must be at least 10.
TIME	CHARACTER**n*	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	CHARACTER**n*	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	not supported	
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	$1 \leq n \leq 2147483647$
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	$1 \leq n \leq 2147483647$
DBCLOB(*n*)	not supported	
ROWID	SQL TYPE IS ROWID	

Notes on FORTRAN variable declaration and usage

You should be aware of the following when you declare FORTRAN variables.

Fortran data types with no SQL equivalent: FORTRAN supports some data types with no SQL equivalent (for example, REAL*16 and COMPLEX). In most cases, you can use FORTRAN statements to convert between the unsupported data types and the data types that SQL allows.

SQL data types with no FORTRAN equivalent: FORTRAN does not provide an equivalent for the decimal data type. To hold the value of such a variable, you can use:

- An integer or floating-point variables, which converts the value. If you choose integer, however, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special-purpose FORTRAN data types: The locator data types are FORTRAN data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator
"Chapter 24. Using stored procedures for client/server processing" on page 525

LOB locators "Chapter 13. Programming for large objects (LOBs)" on page 229

Overflow: Be careful of overflow. For example, if you retrieve an INTEGER column value into a INTEGER*2 host variable and the column value is larger than 32767 or -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHARACTER*70 host variable, the rightmost ten characters of the retrieved string are truncated.

Retrieving a double precision floating-point or decimal column value into a INTEGER*4 host variable removes any fractional value.

Processing Unicode data: Because FORTRAN does not support graphic data types, FORTRAN applications can process only Unicode tables that use UTF-8 encoding.

Notes on syntax differences for constants
You should be aware of the following syntax differences for constants.

Real constants: FORTRAN interprets a string of digits with a decimal point to be a real constant. An SQL statement interprets such a string to be a decimal constant. Therefore, use exponent notation when specifying a real (that is, floating-point) constant in an SQL statement.

Exponent indicators: In FORTRAN, a real (floating-point) constant having a length of eight bytes uses a D as the exponent indicator (for example, 3.14159D+04). An 8-byte floating-point constant in an SQL statement must use an E (for example, 3.14159E+04).

Determining compatibility of SQL and FORTRAN data types

Host variables must be type compatible with the column values with which you intend to use them.

- Numeric data types are compatible with each other. For example, if a column value is INTEGER, you must declare the host variable as INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, or DOUBLE PRECISION.
- Character data types are compatible with each other. A CHAR, VARCHAR, or CLOB column is compatible with FORTRAN character host variable.
- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.
- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a FORTRAN character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB host variable.
- The ROWID column is compatible only with a ROWID host variable.
- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. For information on assigning and comparing distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

Using indicator variables

An indicator variable is a 2-byte integer (INTEGER*2). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see "Using indicator variables with host variables" on page 70.

Example: Given the statement:

```
 EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C                                 :DAY :DAYIND,
C                                 :BGN :BGNIND,
C                                 :END :ENDIND
```

You can declare variables as follows:

```
CHARACTER*7 CLSCD
INTEGER*2   DAY
CHARACTER*8 BGN, END
INTEGER*2   DAYIND, BGNIND, ENDIND
```

The following figure shows the syntax for a valid indicator variable.

```
►►──INTEGER*2──variable-name──────────────────────────────────────────►◄
                            └─/──numeric-constant──/─┘
```
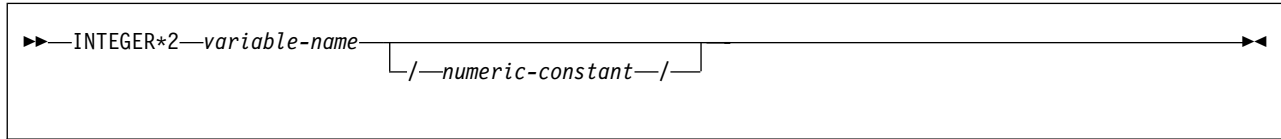
Figure 74. Indicator variable

Handling SQL error return codes

You can use the subroutine DSNTIR to convert an SQL return code into a text message. DSNTIR builds a parameter list and calls DSNTIAR for you. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL error return codes" on page 76.

DSNTIR syntax

CALL DSNTIR (*error-length, message, return-code*)

The DSNTIR parameters have the following meanings:

error-length
    The total length of the message output area.

message
    An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

    The output lines of text are put into this area. For example, you could specify the format of the output area as:

```
INTEGER   ERRLEN /1320/
CHARACTER*132 ERRTXT(10)
INTEGER   ICODE
 :
CALL DSNTIR ( ERRLEN, ERRTXT, ICODE )
```

    where ERRLEN is the total length of the message output area, ERRTXT is the name of the message output area, and ICODE is the return code.

return-code
    Accepts a return code from DSNTIAR.

An example of calling DSNTIR (which then calls DSNTIAR) from an application appears in the DB2 sample assembler program DSN8BF3, contained in the library DSN8710.SDSNSAMP. See "Appendix B. Sample applications" on page 829 for instructions on how to access and print the source code for the sample program.

Coding SQL statements in a PL/I application

This section helps you with the programming techniques that are unique to coding SQL statements within a PL/I program.

Defining the SQL communication area

A PL/I program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as BIN FIXED (31)
- An SQLSTATE variable declared as CHARACTER(5)

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)
When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If you specify STDSQL(NO)
When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a PL/I program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:
- CALL...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
- DESCRIBE INPUT *statement-name* INTO *descriptor-name*
- DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
- DESCRIBE TABLE *host-variable* INTO *descriptor-name*
- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*

- PREPARE...INTO *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. You can code an SQLDA in a PL/I program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

You must declare an SQLDA before the first SQL statement that references that data descriptor, unless you use the precompiler option TWOPASS. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix C of *DB2 SQL Reference* for a complete description of SQLDA fields.

Embedding SQL statements

The first statement of the PL/I program must be the statement PROCEDURE with OPTIONS(MAIN), unless the program is a stored procedure. A stored procedure application can run as a subroutine. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for more information.

You can code SQL statements in a PL/I program wherever you can use executable statements.

Each SQL statement in a PL/I program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear all on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a PL/I program as follows:

```
EXEC SQL  UPDATE DSN8710.DEPT
             SET MGRNO = :MGR_NUM
             WHERE DEPTNO = :INT_DEPT ;
```

Comments: You can include PL/I comments in embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can also include SQL comments in any static SQL statement if you specify the precompiler option STDSQL(YES).

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for other PL/I statements, except that you must specify EXEC SQL on one line.

Declaring tables and views: Your PL/I program should also include a DECLARE TABLE statement to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For details, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

Including code: You can use SQL statements or PL/I host variable declarations from a member of a partitioned data set by using the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use the statement PL/I %INCLUDE to include SQL statements or host variable DCL statements. You must use the PL/I preprocessor to resolve any %INCLUDE statements before you use the DB2 precompiler. Do not use PL/I preprocessor directives within SQL statements.

Margins: Code SQL statements in columns 2 through 72, unless you have specified other margins to the DB2 precompiler. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid PL/I name for a host variable. Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers. IEL0378 messages from the PL/I compiler identify lines of code without sequence numbers. You can ignore these messages.

Statement labels: You can specify a statement label for executable SQL statements. However, the statements INCLUDE *text-file-name* and END DECLARE SECTION cannot have statement labels.

Whenever statement: The target for the GOTO clause in an SQL statement WHENEVER must be a label in the PL/I source code and must be within the scope of any SQL statements that WHENEVER affects.

Using double-byte character set (DBCS) characters: The following considerations apply to using DBCS in PL/I programs with SQL statements:
- If you use DBCS in the PL/I source, then DB2 rules for the following language elements apply:
  - Graphic strings
  - Graphic string constants
  - Host identifiers
  - Mixed data in character strings
  - MIXED DATA option

  See Chapter 2 of *DB2 SQL Reference* for detailed information about language elements.
- The PL/I preprocessor transforms the format of DBCS constants. If you do not want that transformation, run the DB2 precompiler *before* the preprocessor.
- If you use graphic string constants or mixed data in dynamically prepared SQL statements, and if your application requires the PL/I Version 2 compiler, then the dynamically prepared statements must use the PL/I mixed constant format.
  - If you prepare the statement from a host variable, change the string assignment to a PL/I mixed string.
  - If you prepare the statement from a PL/I string, change that to a host variable and then change the string assignment to a PL/I mixed string.

  For example:
  ```
  SQLSTMT = 'SELECT'<dbdb>'' FROM table-name'M;
  EXEC SQL PREPARE FROM :SQLSTMT;
  ```

  For instructions on preparing SQL statements dynamically, see "Chapter 23. Coding dynamic SQL in application programs" on page 495.

- If you want a DBCS identifier to resemble PL/I graphic string, you must use a delimited identifier.
- If you include DBCS characters in comments, you must delimit the characters with a shift-out and shift-in control character. The first shift-in character signals the end of the DBCS string.
- You can declare host variable names that use DBCS characters in PL/I application programs. The rules for using DBCS variable names in PL/I follow existing rules for DBCS SQL Ordinary Identifiers, except for length. The maximum length for a host variable is 64 single-byte characters in DB2. Please see Chapter 2 of *DB2 SQL Reference* for the rules for DBCS SQL Ordinary Identifiers.

  Restrictions:
  – DBCS variable names must contain DBCS characters only. Mixing single-byte character set (SBCS) characters with DBCS characters in a DBCS variable name produces unpredictable results.
  – A DBCS variable name cannot continue to the next line.
- The PL/I preprocessor changes non-Kanji DBCS characters into extended binary coded decimal interchange code (EBCDIC) SBCS characters. To avoid this change, use Kanji DBCS characters for DBCS variable names, or run the PL/I compiler without the PL/I preprocessor.

Special PL/I considerations: The following considerations apply to programs written in PL/I.

- When compiling a PL/I program that includes SQL statements, you must use the PL/I compiler option CHARSET (60 EBCDIC).
- In unusual cases, the generated comments in PL/I can contain a semicolon. The semicolon generates compiler message IEL0239I, which you can ignore.
- The generated code in a PL/I declaration can contain the ADDR function of a field defined as character varying. This produces message IEL0872, which you can ignore.
- The precompiler generated code in PL/I source can contain the NULL() function. This produces message IEL0533I, which you can ignore unless you have also used NULL as a PL/I variable. If you use NULL as a PL/I variable in a DB2 application, then you must also declare NULL as a built-in function (DCL NULL BUILTIN;) to avoid PL/I compiler errors.
- The PL/I macro processor can generate SQL statements or host variable DCL statements if you run the macro processor before running the DB2 precompiler.

  If you use the PL/I macro processor, do not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the COPTION parameter of the DSNH command or the option PARM.PLI=*options* of the EXEC statement in the DSNHPLI procedure.
- Use of the PL/I multitasking facility, where multiple tasks execute SQL statements, causes unpredictable results. See the RUN(DSN) command in Chapter 2 of *DB2 Command Reference*.

Using host variables

You must explicitly declare all host variable before their first use in the SQL statements, unless you specify the precompiler option TWOPASS. If you specify the precompiler option TWOPASS, you must declare the host variables before its use in the statement DECLARE CURSOR.

You can precede PL/I statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement, with the following exception: If the SQL statement meets the following conditions, a host variable in the SQL statement *cannot* be preceded by a colon:
- The SQL statement is an EXECUTE IMMEDIATE or PREPARE statement.
- The SQL statement is in a program that also contains a DECLARE VARIABLE statement.
- The host variable is part of a string expression, but the host variable is not the only component of the string expression.

The names of host variables should be unique within the program, even if the host variables are in different blocks or procedures. You can qualify the host variable names with a structure name to make them unique.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Host variables must be scalar variables or structures of scalars. You cannot declare host variables as arrays, although you can use an array of indicator variables when you associate the array with a host structure.

Declaring host variables

Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults specified in the statement PL/I DEFAULT. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message ″UNDECLARED HOST VARIABLE″.

The precompiler uses only the names and data attributes of the variables; it ignores the alignment, scope, and storage attributes. Even though the precompiler ignores alignment, scope, and storage, if you ignore the restrictions on their use, you might have problems compiling the PL/I source code that the precompiler generates. These restrictions are as follows:

- A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
- If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
- Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations.

```
►►─┬─DECLARE─┬──┬─variable-name────────────────────┬──────────────────►
   └─DCL─────┘  │          ┌─,─────────────┐       │
               │          │  ▼            │        │
               └─(─────────variable-name───┴──)──┘

►─┬─BINARY──┬──┬─FIXED─┬──────────────────────────┬──────────────────────►
  ├─BIN─────┤  │       └─(─precision─┬──────────┬─)─┘
  ├─DECIMAL─┤  │                     └─,scale───┘
  └─DEC─────┘  └─FLOAT──(─precision─)────────────────┘

►─┬──────────────────────────────────────────┬──────────────────────►◄
  └─Alignment and/or Scope and/or Storage──┘
```
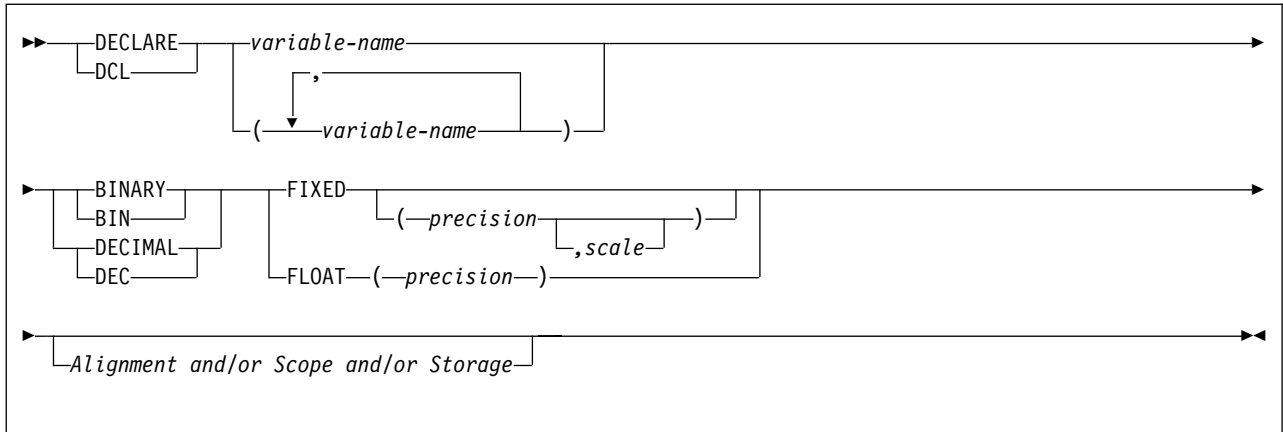
Figure 75. Numeric host variables

Notes:

1. You can specify host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

2. You can specify a *scale* for only DECIMAL FIXED.

Character host variables: The following figure shows the syntax for valid character host variable declarations, other than CLOBs. See Figure 80 on page 180 for the syntax of CLOBs.

```
►►─┬─DECLARE─┬──┬─variable-name──────────────────┬──┬─CHARACTER─┬──(─length─)──┬─────────┬─►
   └─DCL─────┘  │         ┌─,──────────────┐     │  └─CHAR──────┘              ├─VARYING─┤
               │         │  ▼             │      │                            └─VAR─────┘
               └─(────────variable-name───┴──)──┘

►─┬──────────────────────────────────────────┬──────────────────────►◄
  └─Alignment and/or Scope and/or Storage──┘
```
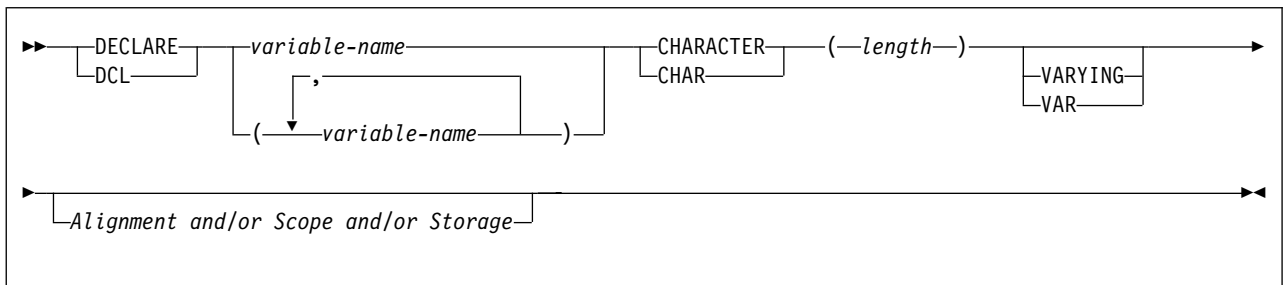
Figure 76. Character host variables

Graphic host variables: The following figure shows the syntax for valid graphic host variable declarations, other than DBCLOBs. See Figure 80 on page 180 for the syntax of DBCLOBs.

```
►►─┬─DECLARE─┬──┬─variable-name──────────────────┬──GRAPHIC──(─length─)──┬─────────┬─►
   └─DCL─────┘  │         ┌─,──────────────┐     │                       ├─VARYING─┤
               │         │  ▼             │      │                       └─VAR─────┘
               └─(────────variable-name───┴──)──┘

►─┬──────────────────────────────────────────┬──────────────────────►◄
  └─Alignment and/or Scope and/or Storage──┘
```
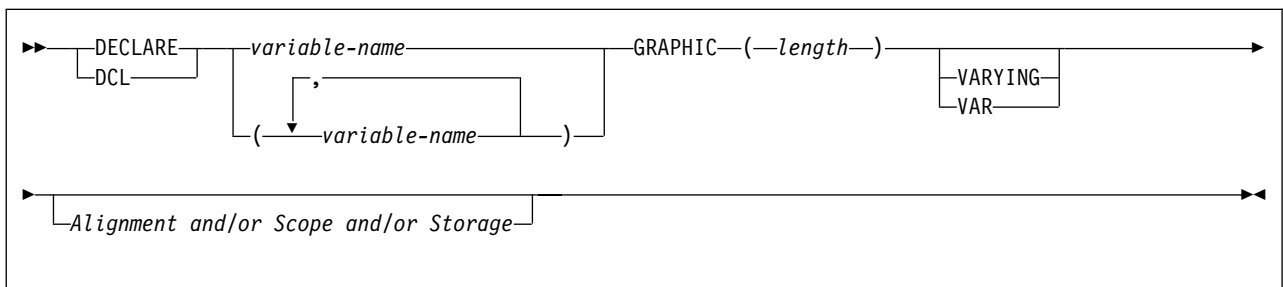
Figure 77. Graphic host variables

Result set locators: The following figure shows the syntax for valid result set locator declarations. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for a discussion of how to use these host variables.

```
►►──┬─DECLARE─┬───┬─variable-name──────────────┬── SQL TYPE IS RESULT_SET_LOCATOR VARYING───────►
    └─DCL─────┘   │         ┌─,──────────┐      │
                  │         ▼            │      │
                  └─(─────────variable-name───)─┘

►──┬──────────────────────────────────────┬───────────────────────────────────────────────►◄
   └─Alignment and/or Scope and/or Storage─┘
```
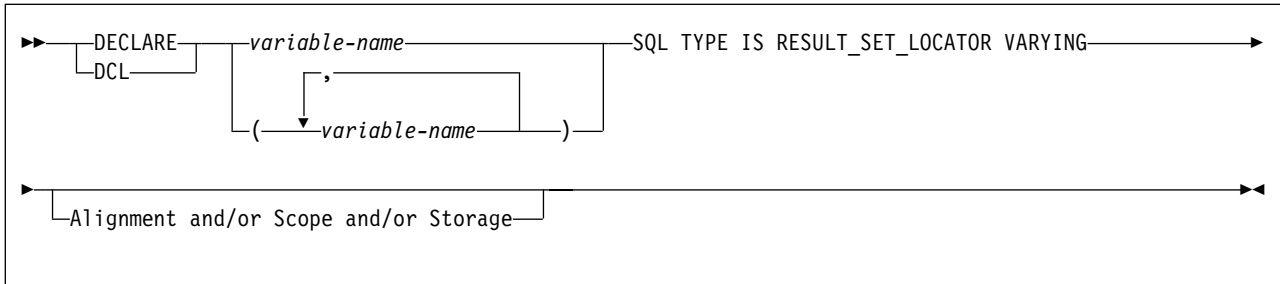
Figure 78. Result set locators

Table Locators: The following figure shows the syntax for declarations of table locators. See "Accessing transition tables in a user-defined function or stored procedure" on page 278 for a discussion of how to use these host variables.

```
►►──┬─DCL─────┬───┬─variable-name──────────────┬── SQL TYPE IS──TABLE LIKE──table-name──AS LOCATOR──►◄
    └─DECLARE─┘   │       ┌─,────────┐          │
                  │       ▼          │          │
                  └─(───────variable-name──)────┘
```
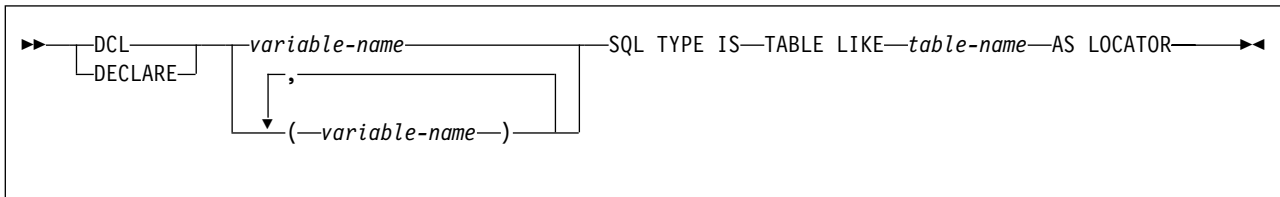
Figure 79. Table locators

LOB Variables and Locators: The following figure shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►──┬─DCL─────┬───┬─variable-name──────────────┬── SQL──TYPE──IS──────────────────────────────────►
    └─DECLARE─┘   │       ┌─,────────┐          │
                  │       ▼          │          │
                  └─(───────variable-name──)────┘

►──┬─┬─BINARY LARGE OBJECT───┬──(──length──┬───┬──)─┬──────────────────────────────────────────►◄
   │ └─BLOB──────────────────┘             ├─K─┤    │
   │ ┌─CHARACTER LARGE OBJECT─┐            ├─M─┤    │
   │ ├─CHAR LARGE OBJECT───────┤           └─G─┘    │
   │ ├─CLOB────────────────────┘                    │
   │ └─DBCLOB──────────────────                      │
   ├─BLOB_LOCATOR───────────────────────────────────┘
   ├─CLOB_LOCATOR───────────
   └─DBCLOB_LOCATOR─────────
```
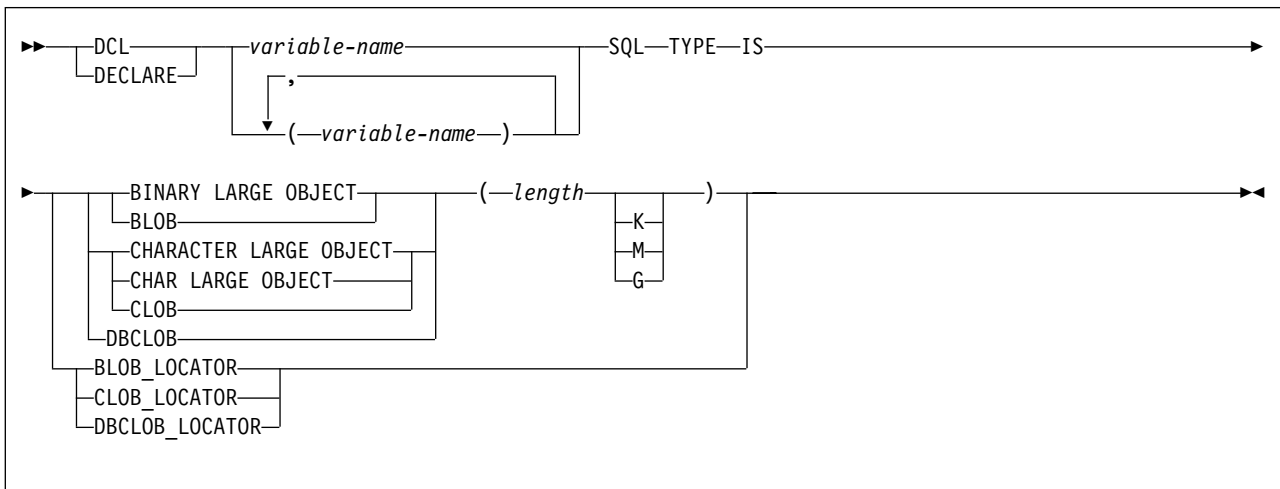
Figure 80. LOB variables and locators

ROWIDs: The following figure shows the syntax for declarations of ROWID variables. See "Chapter 13. Programming for large objects (LOBs)" on page 229 for a discussion of how to use these host variables.

```
►►──┬─DCL─────┬──┬─variable-name─────────────┬──SQL TYPE IS──ROWID──────────────────────────►◄
    └─DECLARE─┘  │          ┌─,─────────┐     │
                 │          ▼            │     │
                 └──(──variable-name──)──┴─────┘
```
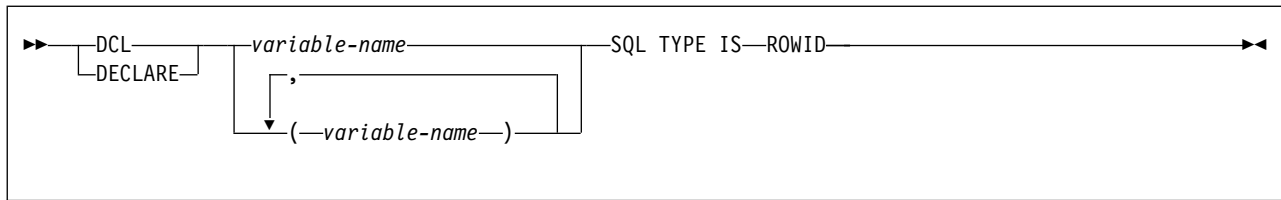
Figure 81. ROWID variables

Using host structures

A PL/I host structure name can be a structure name whose subordinate levels name scalars. For example:

```
DCL 1 A,
      2 B,
        3 C1 CHAR(...),
        3 C2 CHAR(...);
```

In this example, B is the name of a host structure consisting of the scalars C1 and C2.

You can use the structure name as shorthand notation for a list of scalars. You can qualify a host variable with a structure name (for example, STRUCTURE.FIELD). Host structures are limited to two levels. You can think of a host structure for DB2 data as a named group of host variables.

You must terminate the host structure variable by ending the declaration with a semicolon. For example:

```
DCL 1 A,
      2 B CHAR,
      2 (C, D) CHAR;
DCL (E, F) CHAR;
```

You can specify host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following figure shows the syntax for valid host structures.

```
►►──┬─DECLARE─┬──level-1──variable-name─────────────────────┬──,────────────────────────►
    └─DCL─────┘                       └─Scope and/or storage─┘

   ┌─,────────────────────────────────────────┐
   ▼                                           │
►──┴─level-2──┬─var-1──────────────────┬──data-type-specification──┬──;──┬──────────────►◄
              │       ┌─,───┐          │                           └─────┘
              └──(──▼──var-2──┴──)──────┘
```
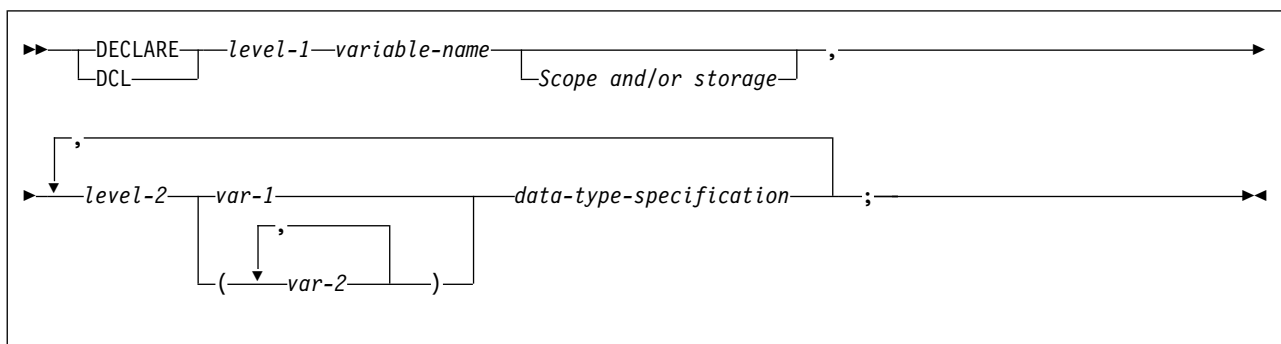
Figure 82. Host structures

PL/I
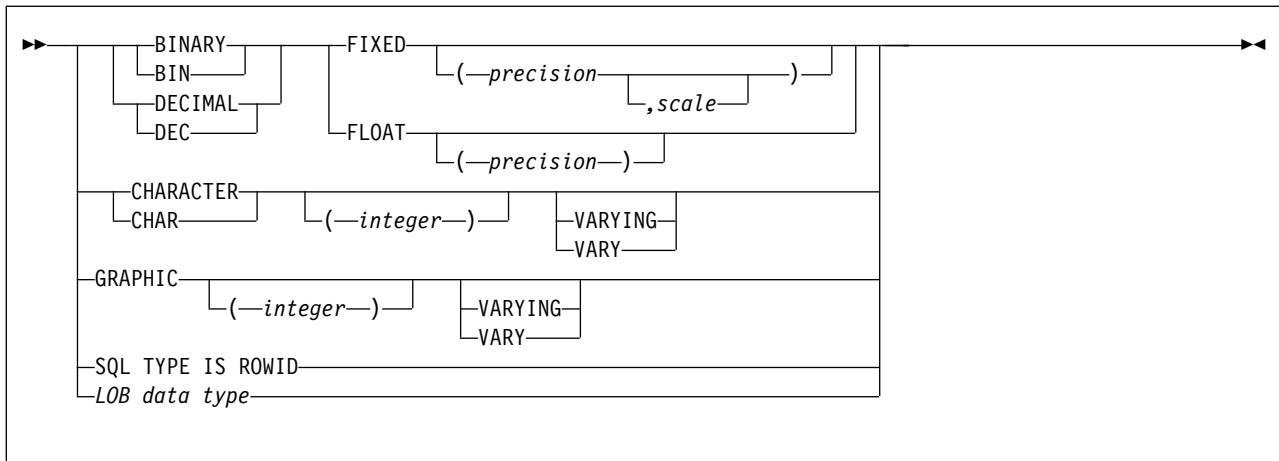
Figure 83. Data type specification
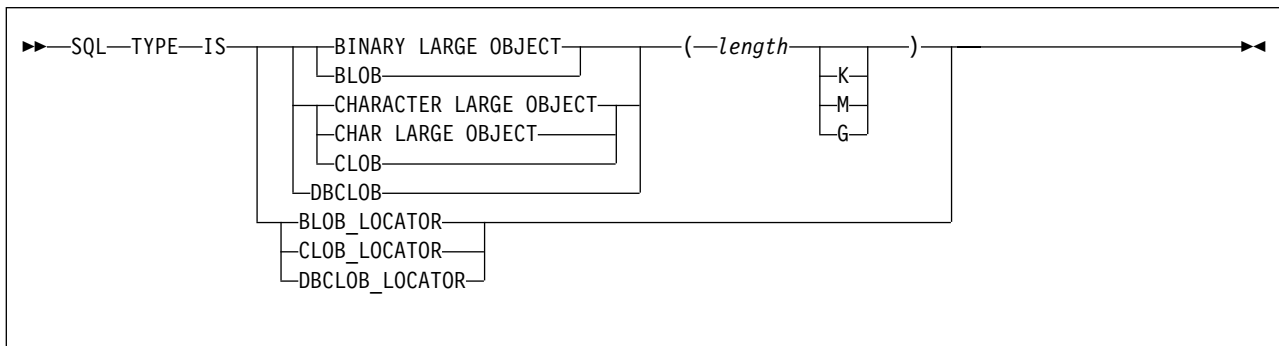
Figure 84. LOB data type

Note: You can specify a *scale* for only DECIMAL FIXED.

Determining equivalent SQL and PL/I data types

Table 16 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 16. SQL data types the precompiler uses for PL/I declarations

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
BIN FIXED(n) $1<=n<=15$	500	2	SMALLINT
BIN FIXED(n) $16<=n<=31$	496	4	INTEGER
DEC FIXED(p,s) $0<=p<=15$ and $0<=s<=p$[1]	484	p in byte 1, s in byte 2	DECIMAL(p,s)
BIN FLOAT(p) $1<=p<=21$	480	4	REAL or FLOAT(n) $1<=n<=21$
BIN FLOAT(p) $22<=p<=53$	480	8	DOUBLE PRECISION or FLOAT(n) $22<=n<=53$
DEC FLOAT(m) $1<=m<=6$	480	4	FLOAT (single precision)

Table 16. SQL data types the precompiler uses for PL/I declarations (continued)

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
DEC FLOAT(*m*) 7<=*m*<=16	480	8	FLOAT (double precision)
CHAR(*n*)	452	*n*	CHAR(*n*)
CHAR(*n*) VARYING 1<=*n*<=255	448	*n*	VARCHAR(*n*)
CHAR(*n*) VARYING *n*>255	456	*n*	VARCHAR(*n*)
GRAPHIC(*n*)	468	*n*	GRAPHIC(*n*)
GRAPHIC(*n*) VARYING 1<=*n*<=127	464	*n*	VARGRAPHIC(*n*)
GRAPHIC(*n*) VARYING *n*>127	472	*n*	VARGRAPHIC(*n*)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator[2]
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table locator[2]
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator[2]
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator[2]
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator[2]
SQL TYPE IS BLOB(*n*) 1≤*n*≤2147483647	404	*n*	BLOB(*n*)
SQL TYPE IS CLOB(*n*) 1≤*n*≤2147483647	408	*n*	CLOB(*n*)
SQL TYPE IS DBCLOB(*n*) 1≤*n*≤1073741823[3]	412	*n*	DBCLOB(*n*)[3]
SQL TYPE IS ROWID	904	40	ROWID

Note:

1. If *p*=0, DB2 interprets it as DECIMAL(15). For example, DB2 interprets a PL/I data type of DEC FIXED(0,0) to be DECIMAL(15,0), which equates to the SQL data type of DECIMAL(15,0).

2. Do not use this data type as a column type.

3. *n* is the number of double-byte characters.

Table 17 on page 184 helps you define host variables that receive output from the database. You can use the table to determine the PL/I data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 17 on page 184 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

PL/I

Table 17. SQL data types mapped to typical PL/I declarations

SQL Data Type	PL/I Equivalent	Notes
SMALLINT	BIN FIXED(n)	$1<=n<=15$
INTEGER	BIN FIXED(n)	$16<=n<=31$

DECIMAL(p,s) or NUMERIC(p,s)	If $p<16$:
	  DEC FIXED(p) or
	  DEC FIXED(p,s)
		p is precision;
		s is scale.
		$1<=p<=31$
		and $0<=s<=p$
		There is no exact equivalent for $p>15$.
		(See 185 for more information.)

REAL or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	$1<=n<=21$, $1<=p<=21$ and $1<=m<=6$
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	$22<=n<=53$, $22<=p<=53$ and $7<=m<=16$
CHAR(n)	CHAR(n)	$1<=n<=255$
VARCHAR(n)	CHAR(n) VAR	
GRAPHIC(n)	GRAPHIC(n)	n refers to the number of double-byte characters, not to the number of bytes. $1<=n<=127$
VARGRAPHIC(n)	GRAPHIC(n) VAR	n refers to the number of double-byte characters, not to the number of bytes.
DATE	CHAR(n)	If you are using a date exit routine, that routine determines n; otherwise, n must be at least 10.
TIME	CHAR(n)	If you are using a time exit routine, that routine determines n. Otherwise, n must be at least 6; to include seconds, n must be at least 8.
TIMESTAMP	CHAR(n)	n must be at least 19. To include microseconds, n must be 26; if n is less than 26, the microseconds part is truncated.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB(n)	SQL TYPE IS BLOB(n)	$1 \le n \le 2147483647$

Table 17. SQL data types mapped to typical PL/I declarations (continued)

SQL Data Type	PL/I Equivalent	Notes
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	1≤n≤2147483647
DBCLOB(*n*	SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. 1≤n≤1073741823
ROWID	SQL TYPE IS ROWID	

Notes on PL/I variable declaration and usage

You should be aware of the following when you declare PL/I variables.

PL/I Data Types with No SQL Equivalent: PL/I supports some data types with no SQL equivalent (COMPLEX and BIT variables, for example). In most cases, you can use PL/I statements to convert between the unsupported PL/I data types and the data types that SQL supports.

SQL data types with no PL/I equivalent: PL/I does not provide an equivalent for the decimal data type when the precision is greater than 15. To hold the value of such a variable, you can use:

- Decimal variables with precision less than or equal to 15, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, then the fractional part of the value could truncate.
- An integer or a floating-point variable, which converts the value. If you choose integer, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special Purpose PL/I Data Types: The locator data types are PL/I data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator
"Chapter 24. Using stored procedures for client/server processing" on page 525

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 278

LOB locators "Chapter 13. Programming for large objects (LOBs)" on page 229

PL/I scoping rules: The precompiler does not support PL/I scoping rules.

Overflow: Be careful of overflow. For example, if you retrieve an INTEGER column value into a BIN FIXED(15) host variable and the column value is larger than 32767 or smaller than -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHAR(70) host variable, the rightmost ten characters of the retrieved string are truncated.

Retrieving a double precision floating-point or decimal column value into a BIN FIXED(31) host variable removes any fractional part of the value.

Similarly, retrieving a DB2 DECIMAL number into a PL/I equivalent variable could truncate the value. This happens because a DB2 DECIMAL value can have up to 31 digits, but a PL/I decimal number can have up to only 15 digits.

Determining compatibility of SQL and PL/I data types

When you use PL/I host variables in SQL statements, the variables must be type compatible with the columns with which you use them.

- Numeric data types are compatible with each other. A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(s,p), BIN FLOAT(n) where n is from 1 to 53, or DEC FLOAT(m) where m is from 1 to 16.
- Character data types are compatible with each other. A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length PL/I character host variable.
- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.
- Graphic data types are compatible with each other. A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length PL/I graphic character host variable.
- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:
  - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column
  - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
  - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.
  - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.
  - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB host variable.

- The ROWID column is compatible only with a ROWID host variable.
- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. For information on assigning and comparing distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables

An indicator variable is a 2-byte integer (BIN FIXED(15)). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see "Using indicator variables with host variables" on page 70.

Example:

Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
                               :DAY :DAY_IND,
                               :BGN :BGN_IND,
                               :END :END_IND;
```

You can declare the variables as follows:

```
DCL CLS_CD    CHAR(7);
DCL DAY       BIN FIXED(15);
DCL BGN       CHAR(8);
DCL END       CHAR(8);
DCL (DAY_IND, BGN_IND, END_IND)   BIN FIXED(15);
```

You can specify host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following figure shows the syntax for a valid indicator variable.
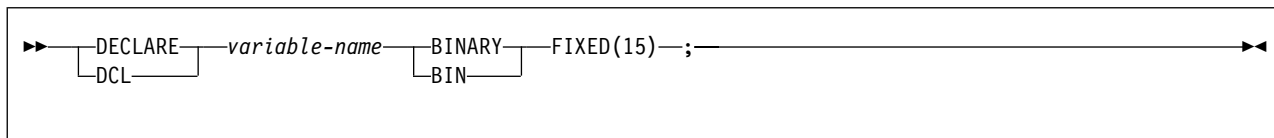
Figure 85. Indicator variable

The following figure shows the syntax for a valid indicator array.

PL/I

```
►►──┬─DECLARE─┬──┬─variable-name──(──dimension──)───────────┬──┬─BINARY─┬──►
    └─DCL─────┘  │                    ┌─,─────────────────┐ │  └─BIN────┘
                 │                    ▼                   │ │
                 └─(────variable-name──(──dimension──)──────┴──)─┘

►─FIXED(15)─┬────────────────────────────────────────┬──;─────────────────►◄
            └─Alignment and/or Scope and/or Storage──┘
```
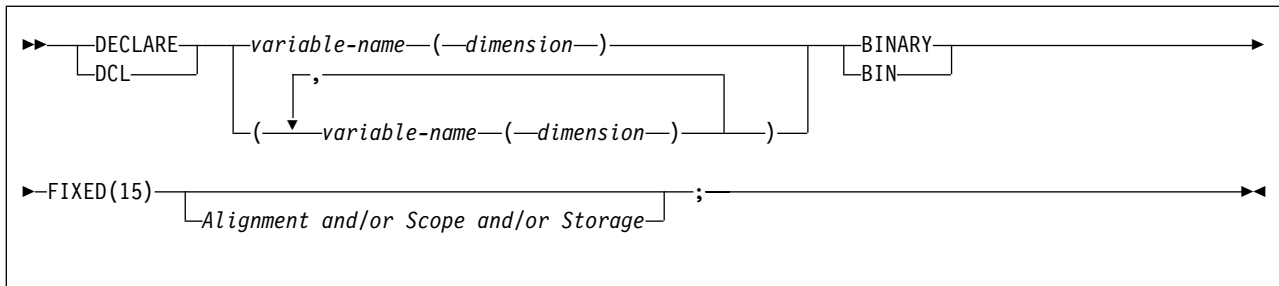
Figure 86. Indicator array

Handling SQL error return codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL error return codes" on page 76.

DSNTIAR syntax

CALL DSNTIAR (*sqlca, message, lrecl*);

The DSNTIAR parameters have the following meanings:

sqlca
    An SQL communication area.

message
    An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

    The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:
```
DCL DATA_LEN FIXED BIN(31) INIT(132);
DCL DATA_DIM FIXED BIN(31) INIT(10);
DCL 1 ERROR_MESSAGE AUTOMATIC,
      3 ERROR_LEN    FIXED BIN(15) UNAL INIT((DATA_LEN*DATA_DIM)),
      3 ERROR_TEXT(DATA_DIM) CHAR(DATA_LEN);
  :
  :
CALL DSNTIAR ( SQLCA, ERROR_MESSAGE, DATA_LEN );
```

    where ERROR_MESSAGE is the name of the message output area, DATA_DIM is the number of lines in the message output area, and DATA_LEN is the length of each line.

lrecl
    A fullword containing the logical record length of output messages, between 72 and 240.

Because DSNTIAR is an assembler language program, you must include the following directives in your PL/I application:
```
DCL DSNTIAR ENTRY OPTIONS (ASM,INTER,RETCODE);
```

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSN8BP3, contained in the library DSN8710.SDSNSAMP. See "Appendix B. Sample applications" on page 829 for instructions on how to access and print the source code for the sample program.

> **CICS**
>
> If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:
>
> ```
> CALL DSNTIAC (eib , commarea , sqlca , msg , lrecl);
> ```
>
> DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.
>
> *eib*   EXEC interface block
>
> *commarea*
>   communication area
>
> For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.
>
> You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.
>
> The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Coding SQL statements in a REXX application

This section helps you with the programming techniques that are unique to coding SQL statements in a REXX procedure. For an example of a complete DB2 REXX procedure, see "Example DB2 REXX application" on page 862.

Defining the SQL communication area

When DB2 prepares a REXX procedure that contains SQL statements, DB2 automatically includes an SQL communication area (SQLCA) in the procedure. The REXX SQLCA differs from the SQLCA for other languages in the following ways:

- The REXX SQLCA consists of a set of separate variables, rather than a structure.

  If you use the `ADDRESS DSNREXX 'CONNECT'` *ssid* syntax to connect to DB2, the SQLCA variables are a set of simple variables.

  If you connect to DB2 using the `CALL SQLDBS 'ATTACH TO'` syntax, the SQLCA variables are compound variables that begin with the stem SQLCA.

  See "Accessing the DB2 REXX Language Support application programming interfaces" on page 190 for a discussion of the methods for connecting a REXX application to DB2.

- You cannot use the INCLUDE SQLCA statement to include an SQLCA in a REXX program.

REXX

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variable values to determine whether the last SQL statement was successful.

See Appendix C of *DB2 SQL Reference* for information on the fields in the REXX SQLCA.

Defining SQL descriptor areas

The following statements require an SQL descriptor area (SQLDA):
- CALL...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
- DESCRIBE INPUT *statement-name* INTO *descriptor-name*
- DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
- DESCRIBE TABLE *host-variable* INTO *descriptor-name*
- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- PREPARE...INTO *descriptor-name*

A REXX procedure can contain more than one SQLDA. Each SQLDA consists of a set of REXX variables with a common stem. The stem must be a REXX variable name that contains no periods and is the same as the value of *descriptor-name* that you specify when you use the SQLDA in an SQL statement. DB2 does not support the INCLUDE SQLDA statement in REXX.

See Appendix C of *DB2 SQL Reference* for information on the fields in a REXX SQLDA.

Accessing the DB2 REXX Language Support application programming interfaces

DB2 REXX Language Support includes the following application programming interfaces:

CONNECT

Connects the REXX procedure to a DB2 subsystem. You must execute CONNECT before you can execute SQL statements. The syntax of CONNECT is:

```
                                 (1)
►►──┬──────────────────┬──'CONNECT'──┬──'subsystem-ID'──┬──────────────────►◄
    └─Address DSNREXX──┘             └─REXX-variable────┘
```
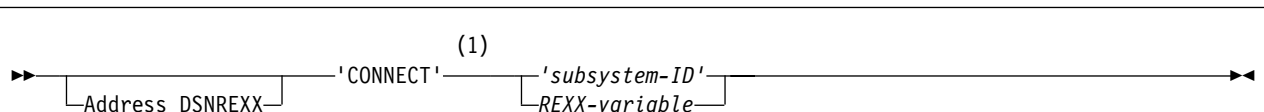
Notes:

1   CALL SQLDBS 'ATTACH TO' *ssid* is equivalent to ADDRESS DSNREXX 'CONNECT' *ssid*.

EXECSQL

Executes SQL statements in REXX procedures. The syntax of EXECSQL is:

```
                                              (1)
►►─────────────────────────"EXECSQL"──────"SQL-statement"──────────────────────►◄
        └Address DSNREXX─┘              └REXX-variable──┘
```

Notes:

1   CALL SQLEXEC is equivalent to EXECSQL.
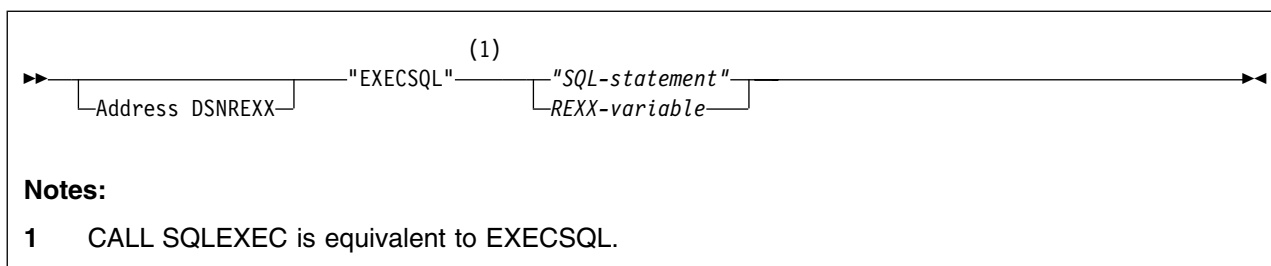
See "Embedding SQL statements in a REXX procedure" on page 192 for more information.

DISCONNECT

Disconnects the REXX procedure from a DB2 subsystem. You should execute DISCONNECT to release resources that are held by DB2. The syntax of DISCONNECT is:

```
                                       (1)
►►──────────────────────'DISCONNECT'──────────────────────────────────────────►◄
        └Address DSNREXX─┘
```

Notes:

1   CALL SQLDBS 'DETACH' is equivalent to DISCONNECT.
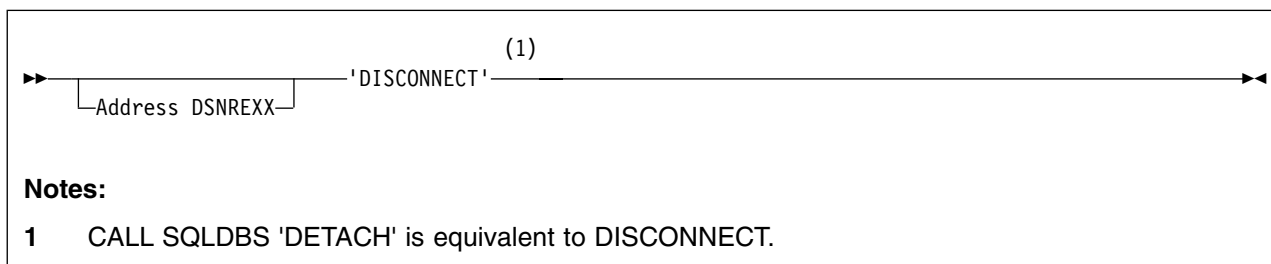
These application programming interfaces are available through the DSNREXX host command environment. To make DSNREXX available to the application, invoke the RXSUBCOM function. The syntax is:

```
►►─RXSUBCOM─(─────'ADD'─────,───'DSNREXX'──,───'DSNREXX'───)───────────────────►◄
             └─'DELETE'─┘
```
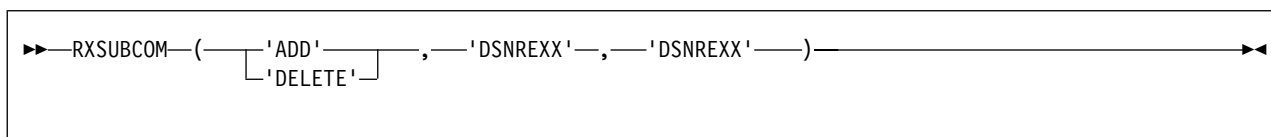
The ADD function adds DSNREXX to the REXX host command environment table. The DELETE function deletes DSNREXX from the REXX host command environment table.

Figure 87 on page 192 shows an example of REXX code that makes DSNREXX available to an application.

```
            'SUBCOM DSNREXX'                     /* HOST CMD ENV AVAILABLE?    */

            IF RC THEN                           /* IF NOT, MAKE IT AVAILABLE  */
              S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
                                                 /* ADD HOST CMD ENVIRONMENT   */

            ADDRESS DSNREXX                      /* SEND ALL COMMANDS OTHER    */
                                                 /* THAN REXX INSTRUCTIONS TO  */
                                                 /* DSNREXX                    */
                                                 /* CALL CONNECT, EXECSQL, AND */
                                                 /* DISCONNECT INTERFACES      */
              .
              .
              .
            S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX')
                                                 /* WHEN DONE WITH             */
                                                 /* DSNREXX, REMOVE IT.        */
```

Figure 87. Making DSNREXX available to an application

Embedding SQL statements in a REXX procedure

You can code SQL statements in a REXX procedure wherever you can use REXX commands. DB2 REXX Language Support allows all SQL statements that DB2 for OS/390 and z/OS supports, *except* the following statements:
- BEGIN DECLARE SECTION
- DECLARE STATEMENT
- END DECLARE SECTION
- INCLUDE
- SELECT INTO
- WHENEVER

Each SQL statement in a REXX procedure must begin with EXECSQL, in either upper-, lower-, or mixed-case. One of the following items must follow EXECSQL:

- An SQL statement enclosed in single or double quotation marks.
- A REXX variable that contains an SQL statement. The REXX variable must not be preceded by a colon.

For example, you can use either of the following methods to execute the COMMIT statement in a REXX procedure:
```
EXECSQL "COMMIT"

rexxvar="COMMIT"
EXECSQL rexxvar
```

You cannot execute a SELECT, INSERT, UPDATE, or DELETE statement that contains host variables. Instead, you must execute PREPARE on the statement, with parameter markers substituted for the host variables, and then use the host variables in an EXECUTE, OPEN, or FETCH statement. See "Using REXX host variables and data types" on page 194 for more information.

An SQL statement follows rules that apply to REXX commands. The SQL statement can optionally end with a semicolon and can be enclosed in single or double quotation marks, as in the following example:
```
'EXECSQL COMMIT';
```

Comments: You cannot include REXX comments (/* ... */) or SQL comments (--) within SQL statements. However, you can include REXX comments anywhere else in the procedure.

Continuation for SQL statements: SQL statements that span lines follow REXX rules for statement continuation. You can break the statement into several strings, each of which fits on a line, and separate the strings with commas or with concatenation operators followed by commas. For example, either of the following statements is valid:

```
EXECSQL ,
   "UPDATE DSN8710.DEPT" ,
   "SET MGRNO = '000010'" ,
   "WHERE DEPTNO = 'D11'"

"EXECSQL " || ,
"  UPDATE DSN8710.DEPT " || ,
"  SET MGRNO = '000010'" || ,
"  WHERE DEPTNO = 'D11'"
```

Including code: The EXECSQL INCLUDE statement is not valid for REXX. You therefore cannot include externally defined SQL statements in a procedure.

Margins: Like REXX commands, SQL statements can begin and end anywhere on a line.

Names: You can use any valid REXX name that does not end with a period as a host variable. However, host variable names should not begin with 'SQL', 'RDI', 'DSN', 'RXSQL', or 'QRW'. Variable names can be at most 64 bytes.

Nulls: A REXX null value and an SQL null value are different. The REXX language has a null string (a string of length 0) and a null clause (a clause that contains only blanks and comments). The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a value. Assigning a REXX null value to a DB2 column does not make the column value null.

Statement labels: You can precede an SQL statement with a label, in the same way that you label REXX commands.

Handling errors and warnings: DB2 does not support the SQL WHENEVER statement in a REXX procedure. To handle SQL errors and warnings, use the following methods:

- To test for SQL errors or warnings, test the SQLCODE or SQLSTATE value and the SQLWARN. values after each EXECSQL call. This method does not detect errors in the REXX interface to DB2.

- To test for SQL errors or warnings or errors or warnings from the REXX interface to DB2, test the REXX RC variable after each EXECSQL call. Table 18 lists the values of the RC variable.

  You can also use the REXX SIGNAL ON ERROR and SIGNAL ON FAILURE keyword instructions to detect negative values of the RC variable and transfer control to an error routine.

Table 18. REXX return codes after SQL statements

Return code	Meaning
0	No SQL warning or error occurred.
+1	An SQL warning occurred.
-1	An SQL error occurred.

Using cursors and statement names

In REXX SQL applications, you must use a predefined set of names for cursors or prepared statements. The following names are valid for cursors and prepared statements in REXX SQL applications:

c1 to c100
Cursor names for DECLARE CURSOR, OPEN, CLOSE, and FETCH statements. By default, c1 to c100 are defined with the WITH RETURN clause, and c51 to c100 are defined with the WITH HOLD clause. You can use the ATTRIBUTES clause of the PREPARE statement to override these attributes or add additional attributes. For example, you might want to add attributes to make your cursor scrollable.

c101 to c200
Cursor names for ALLOCATE, DESCRIBE, FETCH, and CLOSE statements that are used to retrieve result sets in a program that calls a stored procedure.

s1 to s100
Prepared statement names for DECLARE STATEMENT, PREPARE, DESCRIBE, and EXECUTE statements.

Use only the predefined names for cursors and statements. When you associate a cursor name with a statement name in a DECLARE CURSOR statement, the cursor name and the statement must have the same number. For example, if you declare cursor c1, you need to declare it for statement s1:

```
EXECSQL 'DECLARE C1 CURSOR FOR S1'
```

Do not use any of the predefined names as host variables names.

Using REXX host variables and data types

You do not declare host variables in REXX. When you need a new variable, you use it in a REXX command. When you use a REXX variable as a host variable in an SQL statement, you must precede the variable with a colon.

A REXX host variable can be a simple or compound variable. DB2 REXX Language Support evaluates compound variables before DB2 processes SQL statements that contain the variables. In the following example, the host variable that is passed to DB2 is :x.1.2:

```
a=1
b=2
EXECSQL 'OPEN C1 USING :x.a.b'
```

Determining equivalent SQL and REXX data types

All REXX data is string data. Therefore, when a REXX procedure assigns input data to a table column, DB2 converts the data from a string type to the table column type. When a REXX procedure assigns column data to an output variable, DB2 converts the data from the column type to a string type.

When you assign input data to a DB2 table column, you can either let DB2 determine the type that your input data represents, or you can use an SQLDA to tell DB2 the intended type of the input data.

Letting DB2 determine the input data type

You can let DB2 assign a data type to input data based on the format of the input string. Table 19 on page 195 shows the SQL data types that DB2 assigns to input

data and the corresponding formats for that data. The two SQLTYPE values that are listed for each data type are the value for a column that does not accept null values and the value for a column that accepts null values.

If you do not assign a value to a host variable before you assign the host variable to a column, DB2 returns an error code.

Table 19. SQL input data types and REXX data formats

SQL data type assigned by DB2	SQLTYPE for data type	REXX input data format
INTEGER	496/497	A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus (+) or minus (−) sign. The number that is represented must be between -2147483647 and 2147483647, inclusive.
DECIMAL(*p,s*)	484/485	One of the following formats: • A string of numerics that contains a decimal point but no exponent identifier. *p* represents the precision and *s* represents the scale of the decimal number that the string represents. The first character can be a plus (+) or minus (−) sign. • A string of numerics that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (−) sign. The number that is represented is less than -2147483647 or greater than 2147483647.
FLOAT	480/481	A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus (+) or minus (−) sign and a series of numerics). The string can begin with a plus (+) or minus (−) sign.
VARCHAR(*n*)	448/449	One of the following formats: • A string of length *n*, enclosed in single or double quotation marks. • The character X or x, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 2*n bytes and is the hexadecimal representation of a string of *n* characters. • A string of length *n* that does not have a numeric or graphic format, and does not satisfy either of the previous conditions.
VARGRAPHIC(*n*)	464/465	One of the following formats: • The character G, g, N, or n, followed by a string enclosed in single or double quotation marks. The string within the quotation marks begins with a shift-out character (X'0E') and ends with a shift-in character (X'0F'). Between the shift-out character and shift-in character are *n* double-byte characters. • The characters GX, Gx, gX, or gx, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 4*n bytes and is the hexadecimal representation of a string of *n* double-byte characters.

For example, when DB2 executes the following statements to update the MIDINIT column of the EMP table, DB2 must determine a data type for HVMIDINIT:

```
SQLSTMT="UPDATE EMP" ,
   "SET MIDINIT = ?"  ,
   "WHERE EMPNO = '000200'"
```

```
"EXECSQL PREPARE S100 FROM :SQLSTMT"
HVMIDINIT='H'
"EXECSQL EXECUTE S100 USING" ,
   ":HVMIDINIT"
```

Because the data that is assigned to HVMIDINIT has a format that fits a character data type, DB2 REXX Language Support assigns a VARCHAR type to the input data.

Ensuring that DB2 correctly interprets character input data

To ensure that DB2 REXX Language Support does not interpret character literals as graphic or numeric literals, precede and follow character literals with a double quotation mark, followed by a single quotation mark, followed by another double quotation mark ("'").

Enclosing the string in apostrophes is not adequate because REXX removes the apostrophes when it assigns a literal to a variable. For example, suppose that you want to pass the value in host variable stringvar to DB2. The value that you want to pass is the string '100'. The first thing that you need to do is to assign the string to the host variable. You might write a REXX command like this:

```
stringvar = '100'
```

After the command executes, stringvar contains the characters 100 (without the apostrophes). DB2 REXX Language Support then passes the numeric value 100 to DB2, which is not what you intended.

However, suppose that you write the command like this:

```
stringvar = "'"100"'"
```

In this case, REXX assigns the string '100' to stringvar, including the single quotation marks. DB2 REXX Language Support then passes the string '100' to DB2, which is the desired result.

Passing the data type of an input variable to DB2

In some cases, you might want to determine the data type of input data for DB2. For example, DB2 does not assign data types of SMALLINT, CHAR, or GRAPHIC to input data. If you assign or compare this data to columns of type SMALLINT, CHAR, or GRAPHIC, DB2 must do more work than if the data types of the input data and columns match.

To indicate the data type of input data to DB2, use an SQLDA. For example, suppose you want to tell DB2 that the data with which you update the MIDINIT column of the EMP table is of type CHAR, rather than VARCHAR. You need to set up an SQLDA that contains a description of a CHAR column, and then prepare and execute the UPDATE statement using that SQLDA:

```
INSQLDA.SQLD = 1              /* SQLDA contains one variable   */
INSQLDA.1.SQLTYPE = 453       /* Type of the variable is CHAR, */
                              /* and the value can be null     */
INSQLDA.1.SQLLEN  = 1         /* Length of the variable is 1   */
INSQLDA.1.SQLDATA = 'H'       /* Value in variable is H        */
INSQLDA.1.SQLIND  = 0         /* Input variable is not null    */
SQLSTMT="UPDATE EMP" ,
  "SET MIDINIT = ?"  ,
  "WHERE EMPNO = '000200'"
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING" ,
   "DESCRIPTOR :INSQLDA"
```

Retrieving data from DB2 tables

Although all output data is string data, you can determine the data type that the data represents from its format and from the data type of the column from which the data was retrieved. Table 20 gives the format for each type of output data.

Table 20. SQL output data types and REXX data formats

SQL data type	REXX output data format
SMALLINT INTEGER	A string of numerics that does not contain leading zeroes, a decimal point, or an exponent identifier. If the string represents a negative number, it begins with a minus (–) sign. The numeric value is between -2147483647 and 2147483647, inclusive.
DECIMAL(*p,s*)	A string of numerics with one of the following formats: • Contains a decimal point but not an exponent identifier. The string is padded with zeroes to match the scale of the corresponding table column. If the value represents a negative number, it begins with a minus (–) sign. • Does not contain a decimal point or an exponent identifier. The numeric value is less than -2147483647 or greater than 2147483647. If the value is negative, it begins with a minus (–) sign.
FLOAT(*n*) REAL DOUBLE	A string that represents a number in scientific notation. The string consists of a numeric, a decimal point, a series of numerics, and an exponent identifier. The exponent identifier is an E followed by a minus (–) sign and a series of numerics if the number is between -1 and 1. Otherwise, the exponent identifier is an E followed by a series of numerics. If the string represents a negative number, it begins with a minus (–) sign.
CHAR(*n*) VARCHAR(*n*)	A character string of length *n* bytes. The string is not enclosed in single or double quotation marks.
GRAPHIC(*n*) VARGRAPHIC(*n*)	A string of length 2*n* bytes. Each pair of bytes represents a double-byte character. This string does not contain a leading G, is not enclosed in quotation marks, and does not contain shift-out or shift-in characters.

Because you cannot use the SELECT INTO statement in a REXX procedure, to retrieve data from a DB2 table you must prepare a SELECT statement, open a cursor for the prepared statement, and then fetch rows into host variables or an SQLDA using the cursor. The following example demonstrates how you can retrieve data from a DB2 table using an SQLDA:

```
SQLSTMT= ,
'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,' ,
'  WORKDEPT, PHONENO, HIREDATE, JOB,' ,
'  EDLEVEL, SEX, BIRTHDATE, SALARY,' ,
'  BONUS, COMM' ,
'  FROM EMP'
EXECSQL DECLARE C1 CURSOR FOR S1
EXECSQL PREPARE S1 INTO :OUTSQLDA FROM :SQLSTMT
EXECSQL OPEN C1
Do Until(SQLCODE ¬= 0)
  EXECSQL FETCH C1 USING DESCRIPTOR :OUTSQLDA
  If SQLCODE = 0 Then Do
    Line = ''
    Do I = 1 To OUTSQLDA.SQLD
      Line = Line OUTSQLDA.I.SQLDATA
    End I
    Say Line
  End
End
```

Using indicator variables

When you retrieve a null value from a column, DB2 puts a negative value in an indicator variable to indicate that the data in the corresponding host variable is null.

When you pass a null value to DB2, you assign a negative value to an indicator variable to indicate that the corresponding host variable has a null value.

The way that you use indicator variables for input host variables in REXX procedures is slightly different from the way that you use indicator variables in other languages. When you want to pass a null value to a DB2 column, in addition to putting a negative value in an indicator variable, you also need to put a valid value in the corresponding host variable. For example, to set a value of WORKDEPT in table EMP to null, use statements like these:

```
SQLSTMT="UPDATE EMP" ,
  "SET WORKDEPT = ?"
HVWORKDEPT='000'
INDWORKDEPT=-1
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING :HVWORKDEPT :INDWORKDEPT"
```

After you retrieve data from a column that can contain null values, you should always check the indicator variable that corresponds to the output host variable for that column. If the indicator variable value is negative, the retrieved value is null, so you can disregard the value in the host variable.

In the following program, the phone number for employee Haas is selected into variable HVPhone. After the SELECT statement executes, if no phone number for employee Haas is found, indicator variable INDPhone contains -1.

```
'SUBCOM DSNREXX'
IF RC THEN ,
  S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
ADDRESS DSNREXX
'CONNECT' 'DSN'
SQLSTMT = ,
  "SELECT PHONENO FROM DSN8710.EMP WHERE LASTNAME='HAAS'"
"EXECSQL DECLARE C1 CURSOR FOR S1"
"EXECSQL PREPARE S1 FROM :SQLSTMT"
Say "SQLCODE from PREPARE is "SQLCODE
"EXECSQL OPEN C1"
Say "SQLCODE from OPEN is "SQLCODE
"EXECSQL FETCH C1 INTO :HVPhone :INDPhone"
Say "SQLCODE from FETCH is "SQLCODE
If INDPhone < 0 Then ,
Say 'Phone number for Haas is null.'
"EXECSQL CLOSE C1"
Say "SQLCODE from CLOSE is "SQLCODE
S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX')
```

Setting the isolation level of SQL statements in a REXX procedure

When you install DB2 REXX Language Support, you bind four packages for accessing DB2, each with a different isolation level:

Package name	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

To change the isolation level for SQL statements in a REXX procedure, execute the SET CURRENT PACKAGESET statement to select the package with the isolation level you need. For example, to change the isolation level to cursor stability, execute this SQL statement:

```
"EXECSQL SET CURRENT PACKAGESET='DSNREXCS'"
```

REXX

Chapter 10. Using constraints to maintain data integrity

When you modify DB2 tables, you need to ensure that the data is valid. DB2 provides two ways to help you maintain valid data: *constraints* and *triggers*.

Constraints are rules that limit the values that you can insert, delete, or update in a table. There are two types of constraints:

- Table check constraints determine the values that a column can contain. Table check constraints are discussed in "Using table check constraints"
- Referential constraints preserve relationships between tables. Referential constraints are discussed in "Using referential constraints" on page 203.

Triggers are a series of actions that are invoked when a table is updated. Triggers are discussed in "Chapter 11. Using triggers for active data" on page 209.

Using table check constraints

Table check constraints designate the values that specific columns of a base table can contain, providing you a method of controlling the integrity of data entered into tables. You can create tables with table check constraints using the CREATE TABLE statement, or you can add the constraints with the ALTER TABLE statement. However, if the check integrity is compromised or cannot be guaranteed for a table, the table space or partition that contains the table is placed in a check pending state. Check integrity is the condition that exists when each row of a table conforms to the check constraints defined on that table.

For example, you might want to make sure that no salary can be below 15000 dollars:

```
CREATE TABLE EMPSAL
(ID          INTEGER      NOT NULL,
 SALARY      INTEGER      CHECK (SALARY >= 15000));
```

Figure 88. Creating a simple table check constraint

Using table check constraints makes your programming task easier, because you do not need to enforce those constraints within application programs or with a validation routine. Define table check constraints on one or more columns in a table when that table is created or altered.

Constraint considerations

The syntax of a table check constraint is checked when the constraint is defined, but the meaning of the constraint is not checked. The following examples show mistakes that are not caught. Column C1 is defined as INTEGER NOT NULL.

Allowable but mistaken check constraints:

- A self-contradictory check constraint:

  ```
  CHECK (C1 > 5 AND C1 < 2)
  ```
- Two check constraints that contradict each other:

  ```
  CHECK (C1 > 5)
  CHECK (C1 < 2)
  ```
- Two check constraints, one of which is redundant:

```
CHECK (C1 >  0)
CHECK (C1 >= 1)
```
- A check constraint that contradicts the column definition:
  ```
  CHECK (C1 IS NULL)
  ```
- A check constraint that repeats the column definition:
  ```
  CHECK (C1 IS NOT NULL)
  ```

A table check constraint is not checked for consistency with other types of constraints. For example, a column in a dependent table can have a referential constraint with a delete rule of SET NULL. You can also define a check constraint that prohibits nulls in the column. As a result, an attempt to delete a parent row fails, because setting the dependent row to null violates the check constraint.

Similarly, a table check constraint is not checked for consistency with a validation routine, which is applied to a table before a check constraint. If the routine requires a column to be greater than or equal to 10 and a check constraint requires the same column to be less than 10, table inserts are not possible. Plans and packages do not need to be rebound after table check constraints are defined on or removed from a table.

When table check constraints are enforced

After table check constraints are defined on a table, any change must satisfy those constraints if it is made by:
- The LOAD utility with the option ENFORCE CONSTRAINT
- An SQL INSERT statement
- An SQL UPDATE statement

A row satisfies a check constraint if its condition evaluates either to true or to unknown. A condition can evaluate to unknown for a row if one of the named columns contains the null value for that row.

Any constraint defined on columns of a base table applies to the views defined on that base table.

When you use ALTER TABLE to add a table check constraint to already populated tables, the enforcement of the check constraint is determined by the value of the CURRENT RULES special register as follows:
- If the value is STD, the check constraint is enforced immediately when it is defined. If a row does not conform, the table check constraint is not added to the table and an error occurs.
- If the value is DB2, the check constraint is added to the table description but its enforcement is deferred. Because there might be rows in the table that violate the check constraint, the table is placed in check pending status.

How table check constraints set check pending status

Maintaining check integrity requires enforcing check constraints on data in a table. When check integrity is compromised or cannot be guaranteed, the table space or partition that contains the table is placed in check pending status. The definition of that status includes violations of table check constraints as well as referential constraints.

Table check violations place a table space or partition in check pending status when any of these conditions exist:

- A table check constraint is defined on a populated table using the ALTER TABLE statement, and the value of the CURRENT RULES special register is DB2.
- The LOAD utility is run with CONSTRAINTS NO, and table check constraints are defined on the table.
- CHECK DATA is run on a table that contains violations of table check constraints.
- A point-in-time RECOVER introduces violations of table check constraints.

Using referential constraints

A table can serve as the "master list" of all occurrences of an entity. In the sample application, the employee table serves that purpose for employees; the numbers that appear in that table are the only valid employee numbers. Likewise, the department table provides a master list of all valid department numbers; the project activity table provides a master list of activities performed for projects; and so on.
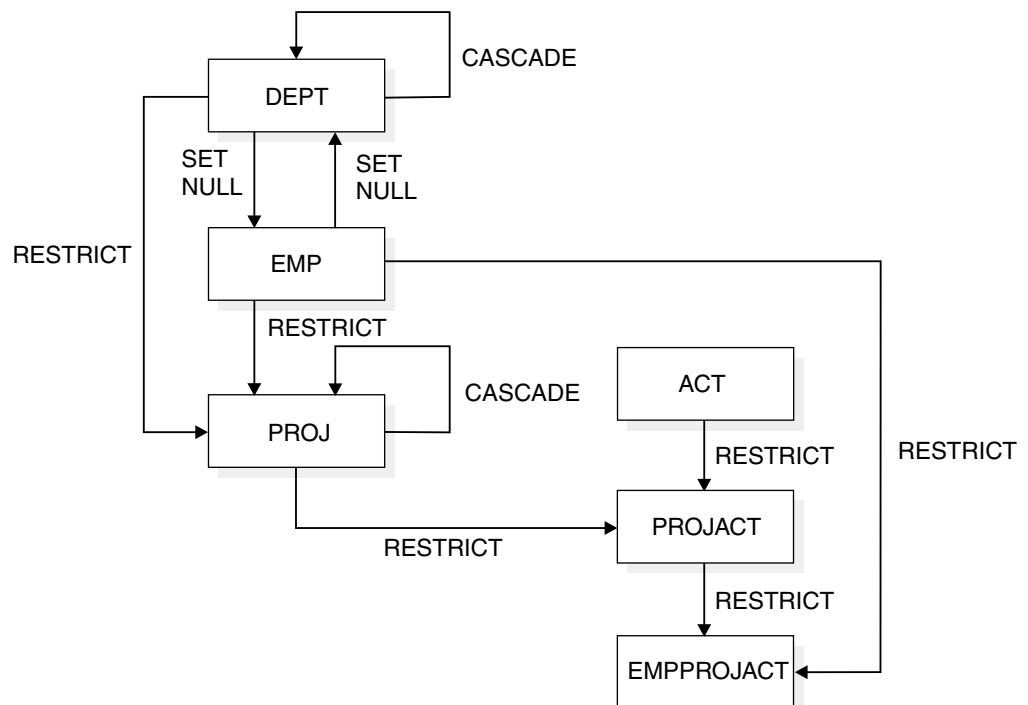
Figure 89. Relationships among tables in the sample application. Arrows point from parent tables to dependent tables.

When a table refers to an entity for which there is a master list, it should identify an occurrence of the entity that actually appears in the master list; otherwise, either the reference is invalid or the master list is incomplete. Referential constraints enforce the relationship between a table and a master list.

Parent key columns

If every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the *parent key* of the table. To ensure that the parent key does not contain duplicate values, you must create a unique index on the column or columns that constitute the parent key. Defining the parent key is called entity integrity, because it requires each entity to have a unique key.

In some cases, using a timestamp as part of the key can be helpful, for example when a table does not have a "natural" unique key or if arrival sequence is the key.

Primary keys for some of the sample tables are:

Table	Key Column
Employee table	EMPNO
Department table	DEPTNO
Project table	PROJNO

Figure 90 shows part of the project table with the primary key column indicated.

Project table

Primary key column

PROJNO	PROJNAME	DEPTNO
MA2100	WELD LINE AUTOMATION	D01
MA2110	W L PROGRAMMING	D11

Figure 90. A primary key on a table

Figure 91 shows a primary key containing more than one column; the primary key is a *composite key*.

Project activity table

Primary key columns

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	.50	1982-01-01	1982-07-01
AD3110	10	1.00	1982-01-01	1983-01-01
AD3111	60	.50	1982-03-15	1982-04-15

Figure 91. A composite primary key. The PROJNO, ACTNO, and ACSTDATE columns are all parts of the key.

Defining a parent key and a unique index

The information in this section (up to "Defining a foreign key" on page 206) is General-use Programming Interface and Associated Guidance Information, as defined in "Notices" on page 937.

The primary key of a table, if one exists, uniquely identifies each occurrence of an entity about which the table contains information. The PRIMARY KEY clause of the CREATE TABLE or ALTER TABLE statements identifies the column or columns of the primary key. Each identified column must be defined as NOT NULL.

Another way to allow only unique values in a column is to create a table using the UNIQUE clause of the CREATE TABLE or ALTER TABLE statement. Like the PRIMARY KEY clause, specifying a UNIQUE clause prevents use of the table until you create an index to enforce the uniqueness of the key. And if you use the UNIQUE clause in an ALTER TABLE statement, a unique index must already exist. For more information about the UNIQUE clause, see Chapter 5 of *DB2 SQL Reference*.

A table that is to be a parent of dependent tables must have a primary or a unique key—the foreign keys of the dependent tables refer to the primary or unique key. Otherwise, a primary key is optional. Consider defining a primary key if each row of your table does pertain to a unique occurrence of some entity. If you define a primary key, an index must be created (the *primary index*) on the same set of columns, in the same order as those columns. If you are defining referential constraints for DB2 to enforce, read "Chapter 10. Using constraints to maintain data integrity" on page 201 before creating or altering any of the tables involved.

A table can have no more than one primary key. A primary key obeys the same restrictions as do index keys:
- The key can include no more than 64 columns.
- No column can be named twice.
- The sum of the column length attributes cannot be greater than 255.

You define a list of columns as the primary key of a table with the PRIMARY KEY clause in the CREATE TABLE statement.

To add a primary key to an existing table, use the PRIMARY KEY clause in an ALTER TABLE statement. In this case, a unique index must already exist.

Incomplete definition
If a table is created with a primary key, its *primary index* is the first unique index created on its primary key columns, with the same order of columns as the primary key columns. The columns of the primary index can be in either ascending or descending order. The table has an incomplete definition until you create an index on the parent key. This incomplete definition status is recorded as a P in the TABLESTATUS column of SYSIBM.SYSTABLES. Use of a table with an incomplete definition is severely restricted: you can drop the table, create the primary index, and drop or create other indexes; you cannot load the table, insert data, retrieve data, update data, delete data, or create foreign keys that reference the primary key.

Because of these restrictions, plan to create the primary index soon after creating the table. For example, to create the primary index for the project activity table, issue:

```
CREATE UNIQUE INDEX XPROJAC1
  ON DSN8710.PROJACT (PROJNO, ACTNO, ACSTDATE);
```

Creating the primary index resets the incomplete definition status and its associated restrictions. But if you drop the primary index, it reverts to incomplete definition status; to reset the status, you must create the primary index or alter the table to drop the primary key.

If the primary key is added later with ALTER TABLE, a unique index on the key columns must already exist. If more than one unique index is on those columns, DB2 chooses one arbitrarily to be the primary index.

Recommendations for defining primary keys
Consider the following items when you plan for primary keys:
- The theoretical model of a relational database suggests that every table should have a primary key to uniquely identify the entities it describes. However, you must weigh that model against the potential cost of index maintenance overhead. DB2 does not require you to define a primary key for tables with no dependents.

- Choose a primary key whose values will not change over time. Choosing a primary key with persistent values enforces the good practice of having unique identifiers that remain the same for the lifetime of the entity occurrence.
- A primary key column should not have default values unless the primary key is a single TIMESTAMP column.
- Choose the minimum number of columns to ensure uniqueness of the primary key.
- A view that can be updated that is defined on a table with a primary key should include all columns of the key. Although this is necessary only if the view is used for inserts, the unique identification of rows can be useful if the view is used for updates, deletes, or selects.
- Drop a primary key later if you change your database or application using SQL.

Defining a foreign key

The information in this section is General-use Programming Interface and Associated Guidance Information, as defined in "Notices" on page 937.

You define a list of columns as a foreign key of a table with the FOREIGN KEY clause in the CREATE TABLE statement.

A foreign key can refer to either a unique or a primary key of the parent table. If the foreign key refers to a non-primary unique key, you must specify the column names of the key explicitly. If the column names of the key are not specified explicitly, the default is to refer to the column names of the primary key of the parent table.

The column names you specify identify the columns of the parent key. The privilege set must include the ALTER or the REFERENCES privilege on the columns of the parent key. A unique index must exist on the parent key columns of the parent table.

The relationship name

You can choose a constraint name (an identifier of up to 8 bytes) for the relationship that is defined by a foreign key. If you do not choose a name, DB2 generates one from the name of the first column of the foreign key, in the same way that it generates the name of an implicitly created table space. For example, the names of the relationships in which the employee-to-project activity table is a dependent would, by default, be recorded (in column RELNAME of SYSIBM.SYSFOREIGNKEYS) as EMPNO and PROJNO. In the following sample CREATE TABLE statement, these constraints are named REPAPA and REPAE.

```
CREATE TABLE DSN8710.EMPPROJACT
      (EMPNO     CHAR(6)          NOT NULL,
       PROJNO    CHAR(6)          NOT NULL,
       ACTNO     SMALLINT         NOT NULL,
      CONSTRAINT REPAPA FOREIGN KEY (PROJNO, ACTNO)
             REFERENCES DSN8710.PROJACT ON DELETE RESTRICT,
      CONSTRAINT REPAE FOREIGN KEY (EMPNO)
             REFERENCES DSN8710.EMP ON DELETE RESTRICT )
   IN DATABASE DSN8D71A;
```

Figure 92. Specifying constraint names for foreign keys

The name is used in error messages, queries to the catalog, and DROP FOREIGN KEY statements. Hence, you might want to choose one if you are experimenting with your database design and have more than one foreign key beginning with the same column (otherwise DB2 generates the name).

Indexes on foreign keys

Although not required, an index on a foreign key is strongly recommended if rows of the parent table are often deleted. The validity of the delete statement, and its possible effect on the dependent table, can be checked through the index.

You can create an index on the columns of a foreign key in the same way you create one on any other set of columns. Most often it is not a unique index. If you do create a unique index on a foreign key, it introduces an additional constraint on the values of the columns.

To let an index on the foreign key be used on the dependent table for a delete operation on a parent table, the leading columns of the index on the foreign key must be identical to and in the same order as the columns in the foreign key.

A foreign key can also be the primary key; then the primary index is also a unique index on the foreign key. In that case, every row of the parent table has at most one dependent row. The dependent table might be used to hold information that pertains to only a few of the occurrences of the entity described by the parent table. For example, a dependent of the employee table might contain information that applies only to employees working in a different country.

The primary key can share columns of the foreign key if the first *n* columns of the foreign key are the same as the primary key's columns. Again, the primary index serves as an index on the foreign key. In the sample project activity table, the primary index (on PROJNO, ACTNO, ACSTDATE) serves as an index on the foreign key on PROJNO. It does not serve as an index on the foreign key on ACTNO, because ACTNO is not the first column of the index.

The FOREIGN KEY clause in ALTER TABLE

You can add a foreign key to an existing table; in fact, that is sometimes the only way to proceed. To make a table self-referencing, you must add a foreign key after creating it.

When a foreign key is added to a populated table, the table space is put into *check pending* status.

Restrictions on cycles of dependent tables

A *cycle* is a set of two or more tables that can be ordered so that each is a dependent of the one before it, and the first is a dependent of the last. Every table in the cycle is a descendent of itself. In the sample application, the employee and department tables are a cycle; each is a dependent of the other.

DB2 does not allow you to create a cycle in which a delete operation on a table involves that same table. Enforcing that principle creates rules about adding a foreign key to a table:
- In a cycle of two tables, neither delete rule can be CASCADE.
- In a cycle of more than two tables, two or more delete rules must not be CASCADE. For example, in a cycle with three tables, two of the delete rules must be other than CASCADE. This concept is illustrated in Figure 93 on page 208.

Alternatively, a delete operation on a self-referencing table must involve the same table, and the delete rule there must be CASCADE or NO ACTION.

Recommendation: Avoid creating a cycle in which all the delete rules are RESTRICT and none of the foreign keys allows nulls. If you do this, no row of any of the tables can ever be deleted.
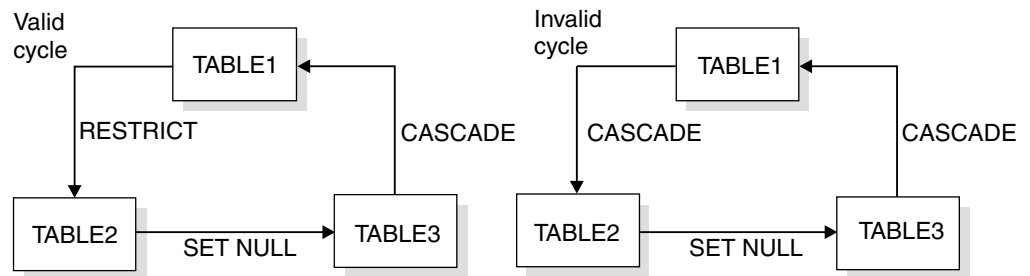
Figure 93. Valid and invalid delete cycles. The left cycle is valid because two or more delete rules are not CASCADE. The cycle on the right is invalid because of the two cascading deletes.

Chapter 11. Using triggers for active data

Triggers are sets of SQL statements that execute when a certain event occurs in a DB2 table. Like constraints, triggers can be used to control changes in DB2 databases. Triggers are more powerful, however, because they can monitor a broader range of changes and perform a broader range of actions than constraints can. For example, a constraint can disallow an update to the salary column of the employee table if the new value is over a certain amount. A trigger can monitor the amount by which the salary changes, as well as the salary value. If the change is above a certain amount, the trigger might substitute a valid value and call a user-defined function to send a notice to an administrator about the invalid update.

Triggers also move application logic into DB2, which can result in faster application development and easier maintenance. For example, you can write applications to control salary changes in the employee table, but each application program that changes the salary column must include logic to check those changes. A better method is to define a trigger that controls changes to the salary column. Then DB2 does the checking for any application that modifies salaries.

This chapter presents the following information about triggers:
- "Example of creating and using a trigger"
- "Parts of a trigger" on page 211
- "Invoking stored procedures and user-defined functions from triggers" on page 217
- "Trigger cascading" on page 218
- "Ordering of multiple triggers" on page 219
- "Interactions among triggers and referential constraints" on page 219
- "Creating triggers to obtain consistent results" on page 221

Example of creating and using a trigger

Triggers automatically execute a set of SQL statements whenever a specified event occurs. These SQL statements can perform tasks such as validation and editing of table changes, reading and modifying tables, or invoking functions or stored procedures that perform operations both inside and outside DB2.

You create triggers using the CREATE TRIGGER statement. Figure 94 on page 210 shows an example of a CREATE TRIGGER statement.

```
                               ▐1▌
CREATE TRIGGER REORDER
      ▐2▌      ▐3▌                        ▐4▌
   AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
                          ▐5▌
   REFERENCING NEW AS N_ROW
           ▐6▌
   FOR EACH ROW MODE DB2SQL
   ▐7▌
   WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
   ▐8▌
   BEGIN ATOMIC
   CALL ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                                   N_ROW.ON_HAND,
                                   N_ROW.PARTNO);
   END
```

Figure 94. Example of a trigger

The parts of this trigger are:

1	Trigger name (REORDER)
2	Trigger activation time (AFTER)
3	Triggering event (UPDATE)
4	Subject table name (PARTS)
5	New transition variable correlation name (N_ROW)
6	Granularity (FOR EACH ROW)
7	Trigger condition (WHEN...)
8	Trigger body (BEGIN ATOMIC...END;)

When you execute this CREATE TRIGGER statement, DB2 creates a trigger
package called REORDER and associates the trigger package with table PARTS.
DB2 records the timestamp when it creates the trigger. If you define other triggers
on the PARTS table, DB2 uses this timestamp to determine which trigger to activate
first. The trigger is now ready to use.

After DB2 updates columns ON_HAND or MAX_STOCKED in any row of table
PARTS, trigger REORDER is activated. The trigger calls a stored procedure called
ISSUE_SHIP_REQUEST if, after a row is updated, the quantity of parts on hand is
less than 10% of the maximum quantity stocked. In the trigger condition, the
qualifier N_ROW represents a value in a modified row after the triggering event.

When you no longer want to use trigger REORDER, you can delete the trigger by
executing the statement:
```
DROP TRIGGER REORDER;
```

Executing this statement drops trigger REORDER and its associated trigger
package named REORDER.

If you drop table PARTS, DB2 also drops trigger REORDER and its trigger
package.

Parts of a trigger

This section gives you the information you need to code each of the trigger parts:
- Trigger name
- Subject table
- Trigger activation time
- Triggering event
- Granularity
- Transition variables
- Transition tables
- Triggered action, which consists of a *trigger condition* and *trigger body*

Trigger name: Use a short, ordinary identifier to name your trigger. You can use a qualifier or let DB2 determine the qualifier. When DB2 creates a trigger package for the trigger, it uses the qualifier for the collection ID of the trigger package. DB2 uses these rules to determine the qualifier:

- If you use static SQL to execute the CREATE TRIGGER statement, DB2 uses the authorization ID in the bind option QUALIFIER for the plan or package that contains the CREATE TRIGGER statement. If the bind command does not include the QUALIFIER option, DB2 uses the owner of the package or plan.
- If you use dynamic SQL to execute the CREATE TRIGGER statement, DB2 uses the authorization ID in special register CURRENT SQLID.

Subject table: When you perform an insert, update, or delete operation on this table, the trigger is activated. You must name a local table in the CREATE TRIGGER statement. You cannot define a trigger on a catalog table or on a view.

Trigger activation time: The two choices for trigger activation time are NO CASCADE BEFORE and AFTER. NO CASCADE BEFORE means that the trigger is activated before DB2 makes any changes to the subject table, and that the triggered action does not activate any other triggers. AFTER means that the trigger is activated after DB2 makes changes to the subject table and can activate other triggers. Triggers with an activation time of NO CASCADE BEFORE are known as before triggers. Triggers with an activation time of AFTER are known as after triggers.

Triggering event: Every trigger is associated with an event. A trigger is activated when the triggering event occurs in the subject table. The triggering event is one of the following SQL operations:
- INSERT
- UPDATE
- DELETE

A triggering event can also be an update or delete operation that occurs as the result of a referential constraint with ON DELETE SET NULL or ON DELETE CASCADE.

Triggers are not activated as the result of updates made to tables by DB2 utilities.

When the triggering event for a trigger is an update operation, the trigger is called an update trigger. Similiarly, triggers for insert operations are called insert triggers, and triggers for delete operations are called delete triggers.

The SQL statement that performs the triggering SQL operation is called the triggering SQL statement.

The following example shows a trigger that is defined with an INSERT triggering event:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END
```

Each triggering event is associated with one subject table and one SQL operation. If the triggering SQL operation is an update operation, the event can be associated with specific columns of the subject table. In this case, the trigger is activated only if the update operation updates any of the specified columns.

For example, the following trigger, PAYROLL1, which invokes user-defined function named PAYROLL_LOG, is activated only if an update operation is performed on columns SALARY or BONUS of table PAYROLL:

```
CREATE TRIGGER PAYROLL1
  AFTER UPDATE OF SALARY, BONUS ON PAYROLL
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE', CURRENT TIME, CURRENT DATE));
  END
```

Granularity: The triggering SQL statement might modify multiple rows in the table. The granularity of the trigger determines whether the trigger is activated only once for the triggering SQL statement or once for every row that the SQL statement modifies. The granularity values are:

- FOR EACH ROW

  The trigger is activated once for each row that DB2 modifies in the subject table. If the triggering SQL statement modifies no rows, the trigger is not activated. However, if the triggering SQL statement updates a value in a row to the same value, the trigger is activated. For example, if an UPDATE trigger is defined on table COMPANY_STATS, the following SQL statement will activate the trigger.

  ```
  UPDATE COMPANY_STATS SET NBEMP = NBEMP;
  ```

- FOR EACH STATEMENT

  The trigger is activated once when the triggering SQL statement executes. The trigger is activated even if the triggering SQL statement modifies no rows.

Triggers with a granularity of FOR EACH ROW are known as row triggers. Triggers with a granularity of FOR EACH STATEMENT are known as statement triggers. Statement triggers can only be after triggers.

The following statement is an example of a row trigger:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END
```

Trigger NEW_HIRE is activated once for every row inserted into the employee table.

Transition variables: When you code a row trigger, you might need to refer to the values of columns in each updated row of the subject table. To do this, specify

transition variables in the REFERENCING clause of your CREATE TRIGGER statement. The two types of transition variables are:

- Old transition variables, specified with the OLD *transition-variable* clause, capture the values of columns before the triggering SQL statement updates them. You can define old transition variables for update and delete triggers.
- New transition variables, specified with the NEW *transition-variable* clause, capture the values of columns after the triggering SQL statement updates them. You can define new transition variables for update and insert triggers.

The following example uses transition variables and invocations of the IDENTITY_VAL_LOCAL function to access values that are assigned to identity columns.

Suppose that you have created tables T and S, with the following definitions:

```
CREATE TABLE T
  (ID SMALLINT GENERATED BY DEFAULT AS IDENTITY (START WITH 100),
   C2 SMALLINT,
   C3 SMALLINT,
   C4 SMALLINT);

CREATE TABLE S
  (ID SMALLINT GENERATED ALWAYS AS IDENTITY,
   C1 SMALLINT);
```

Define a before insert trigger on T that uses the IDENTITY_VAL_LOCAL built-in function to retrieve the current value of identity column ID, and uses transition variables to update the other columns of T with the identity column value.

```
CREATE TRIGGER TR1
  NO CASCADE BEFORE INSERT
  ON T REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.C3 =N.ID;
    SET N.C4 =IDENTITY_VAL_LOCAL();
    SET N.ID =N.C2 *10;
    SET N.C2 =IDENTITY_VAL_LOCAL();
END
```

Now suppose that you execute the following INSERT statement:

```
INSERT INTO S (C1) VALUES (5);
```

This statement inserts a row into S with a value of 5 for column C1 and a value of 1 for identity column ID. Next, suppose that you execute the following SQL statement, which activates trigger TR1:

```
INSERT INTO T (C2)
  VALUES (IDENTITY_VAL_LOCAL());
```

This insert statement, and the subsequent activation of trigger TR1, have the following results:

- The INSERT statement obtains the most recent value that was assigned to an identity column (1), and inserts that value into column C2 of table T. 1 is the value that DB2 inserted into identity column ID of table S.
- When the INSERT statement executes, DB2 inserts the value 100 into identity column ID column of C2.
- The first statement in the body of trigger TR1 inserts the value of transition variable N.ID (100) into column C3. N.ID is the value that identity column ID contains *after* the INSERT statement executes.

- The second statement in the body of trigger TR1 inserts the null value into column C4. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.
- The third statement in the body of trigger TR1 inserts 10 times the value of transition variable N.C2 (10*1) into identity column ID of table T. N.C2 is the value that column C2 contains *after* the INSERT is executed.
- The fourth statement in the body of trigger TR1 inserts the null value into column C2. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.

Transition tables: If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. Like transition variables, transition tables can appear in the REFERENCING clause of a CREATE TRIGGER statement. Transition tables are valid for both row triggers and statement triggers. The two types of transition tables are:

- Old transition tables, specified with the OLD TABLE *transition-table-name* clause, capture the values of columns before the triggering SQL statement updates them. You can define old transition tables for update and delete triggers.
- New transition tables, specified with the NEW TABLE *transition-table-name* clause, capture the values of columns after the triggering SQL statement updates them. You can define new transition variables for update and insert triggers.

The scope of old and new transition table names is the trigger body. If another table exists that has the same name as a transition table, any unqualified reference to that name in the trigger body points to the transition table. To reference the other table in the trigger body, you must use the fully qualified table name.

The following example uses a new transition table to capture the set of rows that are inserted into the INVOICE table:

```
CREATE TRIGGER LRG_ORDR
  AFTER INSERT ON INVOICE
  REFERENCING NEW TABLE AS N_TABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT LARGE_ORDER_ALERT(CUST_NO,
    TOTAL_PRICE, DELIVERY_DATE)
    FROM N_TABLE WHERE TOTAL_PRICE > 10000;
  END
```

The SELECT statement in LRG_ORDER causes user-defined function LARGE_ORDER_ALERT to execute for each row in transition table N_TABLE that satisfies the WHERE clause (TOTAL_PRICE > 10000).

Triggered action: When a trigger is activated, a triggered action occurs. Every trigger has one triggered action, which consists of a trigger condition and a trigger body.

Trigger condition: If you want the triggered action to occur only when certain conditions are true, code a trigger condition. A trigger condition is similar to a predicate in a SELECT, except that the trigger condition begins with WHEN, rather than WHERE. If you do not include a trigger condition in your triggered action, the trigger body executes every time the trigger is activated.

For a row trigger, DB2 evaluates the trigger condition once for each modified row of the subject table. For a statement trigger, DB2 evaluates the trigger condition once for each execution of the triggering SQL statement.

If the trigger condition of a before trigger has a fullselect, the fullselect cannot reference the subject table.

The following example shows a trigger condition that causes the trigger body to execute only when the number of ordered items is greater than the number of available items:

```
CREATE TRIGGER CK_AVAIL
  NO CASCADE BEFORE INSERT ON ORDERS
  REFERENCING NEW AS NEW_ORDER
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_ORDER.QUANTITY >
    (SELECT ON_HAND FROM PARTS
     WHERE NEW_ORDER.PARTNO=PARTS.PARTNO))
     BEGIN ATOMIC
       VALUES(ORDER_ERROR(NEW_ORDER.PARTNO,
         NEW_ORDER.QUANTITY));
     END
```

Trigger body: In the trigger body, you code the SQL statements that you want to execute whenever the trigger condition is true. The trigger body begins with BEGIN ATOMIC and ends with END. You cannot include host variables or parameter markers in your trigger body. If the trigger body contains a WHERE clause that references transition variables, the comparison operator cannot be LIKE.

The statements you can use in a trigger body depend on the activation time of the trigger. Table 21 summarizes which SQL statements you can use in which types of triggers.

Table 21. Valid sql statements for triggers and trigger activation times

SQL Statement	Valid for Activation Time	
	Before	After
SELECT	Yes	Yes
VALUES	Yes	Yes
CALL	Yes	Yes
SIGNAL SQLSTATE	Yes	Yes
SET *transition-variable*	Yes	No
INSERT	No	Yes
UPDATE	No	Yes
DELETE	No	Yes

The following list provides more detailed information about SQL statements that are valid in triggers:

- SELECT, VALUES, and CALL

  Use the SELECT or VALUES statement in a trigger body to conditionally or unconditionally invoke a user-defined function. Use the CALL statement to invoke a stored procedure. See "Invoking stored procedures and user-defined functions from triggers" on page 217 for more information on invoking user-defined functions and stored procedures from triggers.

  A SELECT statement in the trigger body of a before trigger cannot reference the subject table.

- SET *transition-variable*

Because before triggers operate on rows of a table before those rows are modified, you cannot perform operations in the body of a before trigger that directly modify the subject table. You can, however, use the SET *transition-variable* statement to modify the values in a row before those values go into the table. For example, this trigger uses a new transition variable to fill in today's date for the new employee's hire date:

```
CREATE TRIGGER HIREDATE
  NO CASCADE BEFORE INSERT ON EMP
  REFERENCING NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET NEW_VAR.HIRE_DATE = CURRENT_DATE;
  END
```

- SIGNAL SQLSTATE

  Use the SIGNAL SQLSTATE statement in the trigger body to report an error condition and back out any changes that are made by the trigger, as well as actions that result from referential constraints on the subject table. When DB2 executes the SIGNAL SQLSTATE statement, it returns an SQLCA to the application with SQLCODE -438. The SQLCA also includes the following values, which you supply in the SIGNAL SQLSTATE statement:
  - A five-character value that DB2 uses as the SQLSTATE
  - An error message that DB2 places in the SQLERRMC field

  In the following example, the SIGNAL SQLSTATE statement causes DB2 to return an SQLCA with SQLSTATE 75001 and terminate the salary update operation if an employee's salary increase is over 20%:

```
CREATE TRIGGER SAL_ADJ
  BEFORE UPDATE OF SALARY ON EMP
  REFERENCING OLD AS OLD_EMP
  NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY * 1.20))
  BEGIN ATOMIC
    SIGNAL SQLSTATE '75001'
      ('Invalid Salary Increase - Exceeds 20%');
  END
```

- INSERT, UPDATE, and DELETE

  Because you can include INSERT, UPDATE, and DELETE statements in your trigger body, execution of the trigger body might cause activation of other triggers. See "Trigger cascading" on page 218 for more information.

If any SQL statement in the trigger body fails during trigger execution, DB2 rolls back all changes that are made by the triggering SQL statement and the triggered SQL statements. However, if the trigger body executes actions that are outside of DB2's control or are not under the same commit coordination as the DB2 subsystem in which the trigger executes, DB2 cannot undo those actions. Examples of external actions that are not under DB2's control are:
- Performing updates that are not under RRS commit control
- Sending an electronic mail message

If the trigger executes external actions that are under the same commit coordination as the DB2 subsystem under which the trigger executes, and an error occurs during trigger execution, DB2 places the application process that issued the triggering statement in a must-rollback state. The application must then execute a rollback operation to roll back those external actions. Examples of external actions that are under the same commit coordination as the triggering SQL operation are:
- Executing a distributed update operation

- From a user-defined function or stored procedure, executing an external action that affects an external resource manager that is under RRS commit control.

Invoking stored procedures and user-defined functions from triggers

A trigger body can include only SQL statements and built-in functions. Therefore, if you want the trigger to perform actions or use logic that is not available in SQL statements or built-in functions, you need to write a user-defined function or stored procedure and invoke that function or stored procedure from the trigger body. "Chapter 14. Creating and using user-defined functions" on page 241 and "Chapter 24. Using stored procedures for client/server processing" on page 525 contain detailed information on how to write and prepare user-defined functions and stored procedures.

Because a before trigger must not modify any table, functions and procedures that you invoke from a trigger cannot include INSERT, UPDATE, or DELETE statements that modify the subject table.

To invoke a user-defined function from a trigger, code a SELECT statement or VALUES statement. Use a SELECT statement to execute the function conditionally. The number of times the user-defined function executes depends on the number of rows in the result table of the SELECT statement. For example, in this trigger, the SELECT statement causes user-defined function LARGE_ORDER_ALERT to execute for each row in transition table N_TABLE with an order of more than 10000:

```
CREATE TRIGGER LRG_ORDR
  AFTER INSERT ON INVOICE
  REFERENCING NEW TABLE AS N_TABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT LARGE_ORDER_ALERT(CUST_NO, TOTAL_PRICE, DELIVERY_DATE)
      FROM N_TABLE WHERE TOTAL_PRICE > 10000;
  END
```

Use the VALUES statement to execute a function unconditionally; that is, once for each execution of a statement trigger or once for each row in a row trigger. In this example, user-defined function PAYROLL_LOG executes every time an update operation occurs that activates trigger PAYROLL1:

```
CREATE TRIGGER PAYROLL1
  AFTER UPDATE ON PAYROLL
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE',
      CURRENT TIME, CURRENT DATE));
  END
```

To invoke a stored procedure from a trigger, use a CALL statement. The parameters of this stored procedure call must be literals, transition variables, table locators, or expressions.

Passing transition tables to user-defined functions and stored procedures

When you call a user-defined function or stored procedure from a trigger, you might want to give the function or procedure access to the entire set of modified rows. That is, you want to pass a pointer to the old or new transition table. You do this using table locators.

Most of the code for using a table locator is in the function or stored procedure that receives the locator. "Accessing transition tables in a user-defined function or stored procedure" on page 278 explains how a function defines a table locator and uses it to receive a transition table. To pass the transition table from a trigger, specify the parameter TABLE *transition-table-name* when you invoke the function or stored procedure. This causes DB2 to pass a table locator for the transition table to the user-defined function or stored procedure. For example, this trigger passes a table locator for a transition table NEWEMPS to stored procedure CHECKEMP:

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    CALL (CHECKEMP(TABLE NEWEMPS));
  END
```

Trigger cascading

An SQL operation that a trigger performs might modify the subject table or other tables with triggers, so DB2 also activates those triggers. A trigger that is activated as the result of another trigger can be activated at the same level as the original trigger or at a different level. Two triggers, A and B, are activated at different levels if trigger B is activated after trigger A is activated and completes before trigger A completes. If trigger B is activated after trigger A is activated and completes after trigger A completes, then the triggers are at the same level.

For example, in these cases, trigger A and trigger B are activated at the same level:
- Table X has two triggers that are defined on it, A and B. A is a before trigger and B is an after trigger. An update to table X causes both trigger A and trigger B to activate.
- Trigger A updates table X, which has a referential constraint with table Y, which has trigger B defined on it. The referential constraint causes table Y to be updated, which activates trigger B.

In these cases, trigger A and trigger B are activated at different levels:
- Trigger A is defined on table X, and trigger B is defined on table Y. Trigger B is an update trigger. An update to table X activates trigger A, which contains an UPDATE statement on table B in its trigger body. This UPDATE statement activates trigger B.
- Trigger A calls a stored procedure. The stored procedure contains an INSERT statement for table X, which has insert trigger B defined on it. When the INSERT statement on table X executes, trigger B is activated.

When triggers are activated at different levels, it is called *trigger cascading.* Trigger cascading can occur only for after triggers because DB2 does not support cascading of before triggers.

To prevent the possibility of endless trigger cascading, DB2 supports only 16 levels of cascading of triggers, stored procedures, and user-defined functions. If a trigger, user-defined function, or stored procedure at the 17th level is activated, DB2 returns SQLCODE -724 and backs out all SQL changes in the 16 levels of cascading. However, as with any other SQL error that occurs during trigger execution, if any action occurs that is outside the control of DB2, that action is not backed out.

You can write a monitor program that issues IFI READS requests to collect DB2 trace information about the levels of cascading of triggers, user-defined functions,

and stored procedures in your programs. See Appendixes (Volume 2) of *DB2 Administration Guide* for information on how to write a monitor program.

Ordering of multiple triggers

You can create multiple triggers for the same subject table, event, and activation time. The order in which those triggers are activated is the order in which the triggers were created. DB2 records the timestamp when each CREATE TRIGGER statement executes. When an event occurs in a table that activates more than one trigger, DB2 uses the stored timestamps to determine which trigger to activate first.

DB2 always activates all before triggers that are defined on a table before the after triggers that are defined on that table, but within the set of before triggers, the activation order is by timestamp, and within the set of after triggers, the activation order is by timestamp.

In this example, triggers NEWHIRE1 and NEWHIRE2 have the same triggering event (INSERT), the same subject table (EMP), and the same activation time (AFTER). Suppose that the CREATE TRIGGER statement for NEWHIRE1 is run before the CREATE TRIGGER statement for NEWHIRE2:

```
CREATE TRIGGER NEWHIRE1
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END

CREATE TRIGGER NEWHIRE2
  AFTER INSERT ON EMP
  REFERENCING NEW AS N_EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE DEPTS SET NBEMP = NBEMP + 1
      WHERE DEPT_ID = N_EMP.DEPT_ID;
  END
```

When an insert operation occurs on table EMP, DB2 activates NEWHIRE1 first because NEWHIRE1 was created first. Now suppose that someone drops and recreates NEWHIRE1. NEWHIRE1 now has a later timestamp than NEWHIRE2, so the next time an insert operation occurs on EMP, NEWHIRE2 is activated before NEWHIRE1.

If two row triggers are defined for the same action, the trigger that was created earlier is activated first for all affected rows. Then the second trigger is activated for all affected rows. In the previous example, suppose that an INSERT statement with a fullselect inserts 10 rows into table EMP. NEWHIRE1 is activated for all 10 rows, then NEWHIRE2 is activated for all 10 rows.

Interactions among triggers and referential constraints

When you create triggers, you need to understand the interactions among the triggers and constraints on your tables and the effect that the order of processing of those constraints and triggers can have on the results.

In general, the following steps occur when triggering SQL statement S1 performs an insert, update, or delete operation on table T1:

1. DB2 determines the rows of T1 to modify. Call that set of rows M1. The contents of M1 depend on the SQL operation:

- For a delete operation, all rows that satisfy the search condition of the statement for a searched delete operation, or the current row for a positioned delete operation
- For an insert operation, the row identified by the VALUES statement, or the rows identified by the result table of a SELECT clause within the INSERT statement
- For an update operation, all rows that satisfy the search condition of the statement for a searched update operation, or the current row for a positioned update operation

2. DB2 processes all before triggers that are defined on T1, in order of creation.

   Each before trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.

   If an error occurs when the triggered action executes, DB2 rolls back all changes that are made by S1.

3. DB2 makes the changes that are specified in statement S1 to table T1.

   If an error occurs, DB2 rolls back all changes that are made by S1.

4. If M1 is not empty, DB2 applies all the following contraints and checks that are defined on table T1:
   - Referential constraints
   - Check constraints
   - Checks that are due to updates of the table through views defined WITH CHECK OPTION

   Application of referential constraints with rules of DELETE CASCADE or DELETE SET NULL are activated before delete triggers or before update triggers on the dependent tables.

   If any constraint is violated, DB2 rolls back all changes that are made by constraint actions or by statement S1.

5. DB2 processes all after triggers that are defined on T1, and all after triggers on tables that are modified as the result of referential constraint actions, in order of creation.

   Each after row trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.

   Each after statement trigger executes the triggered action once for each execution of S1, even if M1 is empty.

If any triggered actions contain SQL insert, update, or delete operations, DB2 repeats steps 1 through 5 for each operation.

If an error occurs when the triggered action executes, or if a triggered action is at the 17th level of trigger cascading, DB2 rolls back all changes that are made in step 5 and all previous steps.

For example, table DEPT is a parent table of EMP, with these conditions:
- The DEPTNO column of DEPT is the primary key.
- The WORKDEPT column of EMP is the foreign key.
- The constraint is ON DELETE SET NULL.

Suppose the following trigger is defined on EMP:

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
```

```
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  VALUES(CHECKEMP(TABLE NEWEMPS));
END
```

Also suppose that an SQL statement deletes the row with department number E21 from DEPT. Because of the constraint, DB2 finds the rows in EMP with a WORKDEPT value of E21 and sets WORKDEPT in those rows to null. This is equivalent to an update operation on EMP, which has update trigger EMPRAISE. Therefore, because EMPRAISE is an after trigger, EMPRAISE is activated after the constraint action sets WORKDEPT values to null.

Creating triggers to obtain consistent results

When you create triggers and write SQL statements that activate those triggers, you need to ensure that executing those statements on the same set of data always produces the same results. Two common reasons that you can get inconsistent results are:

- Positioned UPDATE or DELETE statements that use uncorrelated subqueries cause triggers to operate on a larger result table than you intended.
- DB2 does not always process rows in the same order, so triggers that propagate rows of a table can generate different result tables at different times.

The following examples demonstrate these situations.

Example: Effect of an uncorrelated subquery on a triggered action: Suppose that tables T1 and T2 look like this:

```
Table T1      Table T2
  A1            B1
  ==            ==
   1             1
   2             2
```

The following trigger is defined on T1:

```
CREATE TRIGGER TR1
  AFTER UPDATE OF T1
  FOR EACH ROW
  MODE DB2SQL
    BEGIN ATOMIC
      DELETE FROM T2 WHERE B1 = 2;
    END
```

Now suppose that an application executes the following statements to perform a positioned update operation:

```
EXEC SQL BEGIN DECLARE SECTION;
 long hv1;
EXEC SQL END DECLARE SECTION;
 :
 :
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT A1 FROM T1
  WHERE A1 IN (SELECT B1 FROM T2)
  FOR UPDATE OF A1;
 :
 :
EXEC SQL OPEN C1;
 :
 :
while(SQLCODE>=0 && SQLCODE!=100)
{
  EXEC SQL FETCH C1 INTO :hv1;
  UPDATE T1 SET A1=5 WHERE CURRENT OF C1;
}
```

When DB2 executes the FETCH statement that positions cursor C1 for the first time, DB2 evaluates the subselect, SELECT B1 FROM T2, to produce a result table that contains the two rows of column T2:

```
1
2
```

When DB2 executes the positioned UPDATE statement for the first time, trigger TR1 is activated. When the body of trigger TR1 executes, the row with value 2 is deleted from T2. However, because SELECT B1 FROM T2 is evaluated only once, when the FETCH statement is executed again, DB2 finds the second row of T1, even though the second row of T2 was deleted. The FETCH statement positions the cursor to the second row of T1, and the second row of T1 is updated. The update operation causes the trigger to be activated again, which causes DB2 to attempt to delete the second row of T2, even though that row was already deleted.

To avoid processing of the second row after it should have been deleted, use a correlated subquery in the cursor declaration:

```
DCL C1 CURSOR FOR
  SELECT A1 FROM T1 X
  WHERE EXISTS (SELECT B1 FROM T2 WHERE X.A1 = B1)
  FOR UPDATE OF A1;
```

In this case, the subquery, SELECT B1 FROM T2 WHERE X.A1 = B1, is evaluated for each FETCH statement. The first time that the FETCH statement executes, it positions the cursor to the first row of T1. The positioned UPDATE operation activates the trigger, which deletes the second row of T2. Therefore, when the FETCH statement executes again, no row is selected, so no update operation or triggered action occurs.

Example: Effect of row processing order on a triggered action: The following example shows how the order of processing rows can change the outcome of an after row trigger.

Suppose that tables T1, T2, and T3 look like this:

```
Table T1     Table T2     Table T3
   A1           B1           C1
   ==           ==           ==
   1          (empty)      (empty)
   2
```

The following trigger is defined on T1:

```
CREATE TRIGGER TR1
  AFTER UPDATE ON T1
  REFERENCING NEW AS N
  FOR EACH ROW
  MODE DB2SQL
    BEGIN ATOMIC
      INSERT INTO T2 VALUES(N.C1);
      INSERT INTO T3 (SELECT B1 FROM T2);
    END
```

Now suppose that a program executes the following UPDATE statement:

```
UPDATE T1 SET A1 = A1 + 1;
```

The contents of tables T2 and T3 after the UPDATE statement executes depend on the order in which DB2 updates the rows of T1.

If DB2 updates the first row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

```
Table T1      Table T2      Table T3
   A1            B1            C1
   ==            ==            ==
    2             2             2
    2
```

After the second row of T1 is updated, the values in the three tables are:

```
Table T1      Table T2      Table T3
   A1            B1            C1
   ==            ==            ==
    2             2             2
    3             3             2
                                3
```

However, if DB2 updates the second row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

```
Table T1      Table T2      Table T3
   A1            B1            C1
   ==            ==            ==
    1             3             3
    3
```

After the first row of T1 is updated, the values in the three tables are:

```
Table T1      Table T2      Table T3
   A1            B1            C1
   ==            ==            ==
    2             3             3
    3             2             3
                                2
```

Part 3. Using DB2 object-relational extensions

Chapter 12. Introduction to DB2 object-relational extensions

With the object extensions of DB2, you can incorporate object-oriented concepts and methodologies into your relational database by extending DB2 with richer sets of data types and functions. With those extensions, you can store instances of object-oriented data types in columns of tables and operate on them using functions in SQL statements. In addition, you can control the types of operations that users can perform on those data types.

The object extensions that DB2 provides are:
- Large objects (LOBs)

  The VARCHAR and VARGRAPHIC data types have a storage limit of 32 KB. Although this might be sufficient for small- to medium-size text data, applications often need to store large text documents. They might also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. DB2 provides three data types to store these data objects as strings of up to 2 GB - 1 in size. The three data types are binary large objects (BLOBs), character large objects (CLOBs), and double-byte character large objects (DBCLOBs).

  For a detailed discussion of LOBs, see "Chapter 13. Programming for large objects (LOBs)" on page 229.

- Distinct types

  A distinct type is a user-defined data type that shares its internal representation with a built-in data type but is considered to be a separate and incompatible type for semantic purposes. For example, you might want to define a picture type or an audio type, both of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

  For a detailed discussion of distinct types, see "Chapter 15. Creating and using distinct types" on page 299.

- User-defined functions

  The built-in functions that are supplied with DB2 are a useful set of functions, but they might not satisfy all of your requirements. For those cases, you can use user-defined functions. For example, a built-in function might perform a calculation you need, but the function does not accept the distinct types you want to pass to it. You can then define a function based on a built-in function, called a *sourced* user-defined function, that accepts your distinct types. You might need to perform another calculation in your SQL statements for which there is no built-in function. In that situation, you can define and write an *external* user-defined function.

  For a detailed discussion of user-defined functions, see "Chapter 14. Creating and using user-defined functions" on page 241.

Chapter 13. Programming for large objects (LOBs)

The term *large object* and the acronym *LOB* refer to DB2 objects that you can use to store large amounts of data. A LOB is a varying-length character string that can contain up to 2 GB - 1 of data.

The three LOB data types are:
* *Binary large object (BLOB)*

  Use a BLOB to store binary data such as pictures, voice, and mixed media.
* *Character large object (CLOB)*

  Use a CLOB to store SBCS or mixed character data, such as documents.
* *Double-byte character large object (DBCLOB)*

  Use a DBCLOB to store data that consists of only DBCS data.

This chapter presents the following information about LOBs:
* "Introduction to LOBs"
* "Declaring LOB host variables and LOB locators" on page 232
* "LOB materialization" on page 236
* "Using LOB locators to save storage" on page 236

Introduction to LOBs

Working with LOBs involves defining the LOBs to DB2, moving the LOB data into DB2 tables, then using SQL operations to manipulate the data. This chapter concentrates on manipulating LOB data using SQL statements. For information on defining LOBs to DB2, see Chapter 5 of *DB2 SQL Reference*. For information on how DB2 utilities manipulate LOB data, see Part 2 of *DB2 Utility Guide and Reference*.

These are the basic steps for defining LOBs and moving the data into DB2:
1. Define a column of the appropriate LOB type and a row identifier (ROWID) column in a DB2 table. Define only one ROWID column, even if there are multiple LOB columns in the table.

   The LOB column holds information about the LOB, not the LOB data itself. The table that contains the LOB information is called the *base table*. DB2 uses the ROWID column to locate your LOB data. You need only one ROWID column in a table that contains one or more LOB columns. You can define the LOB column and the ROWID column in a CREATE TABLE or ALTER TABLE statement. If you are adding a LOB column and a ROWID column to an existing table, you must use two ALTER TABLE statements. Add the ROWID with the first ALTER TABLE statement and the LOB column with the second.
2. Create a table space and table to hold the LOB data.

   The table space and table are called a LOB table space and an auxiliary table. If your base table is nonpartitioned, you must create one LOB table space and one auxiliary table for each LOB column. If your base table is partitioned, for each LOB column, you must create one LOB table space and one auxiliary table for each partition. For example, if your base table has three partitions, you must create three LOB table spaces and three auxiliary tables for each LOB column. Create these objects using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.
3. Create an index on the auxiliary table.

Each auxiliary table must have exactly one index. Use CREATE INDEX for this task.

4. Put the LOB data into DB2.

   If the total length of a LOB column and the base table row is less than 32 KB, you can use the LOAD utility to put the data in DB2. Otherwise, you must use INSERT or UPDATE statements. Even though the data is stored in the auxiliary table, the LOAD utility statement or INSERT statement specifies the base table. Using INSERT can be difficult because your application needs enough storage to hold the entire value that goes into the LOB column.

For example, suppose you want to add a resume for each employee to the employee table. Employee resumes are no more than 5 MB in size. The employee resumes contain single-byte characters, so you can define the resumes to DB2 as CLOBs. You therefore need to add a column of data type CLOB with a length of 5 MB to the employee table. If a ROWID column has not been defined in the table, you need to add the ROWID column before you add the CLOB column. Execute an ALTER TABLE statement to add the ROWID column, and then execute another ALTER TABLE statement to add the CLOB column. You might use statements like this:

```
ALTER TABLE EMP
  ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;
COMMIT;
ALTER TABLE EMP
  ADD EMP_RESUME CLOB(1M);
COMMIT;
```

Next, you need to define a LOB table space and an auxiliary table to hold the employee resumes. You also need to define an index on the auxiliary table. You must define the LOB table space in the same database as the associated base table. You can use statements like this:

```
CREATE LOB TABLESPACE RESUMETS
  IN DSN8D71A
  LOG NO;
COMMIT;
CREATE AUXILIARY TABLE EMP_RESUME_TAB
  IN DSN8D71A.RESUMETS
  STORES DSN8710.EMP
  COLUMN EMP_RESUME;
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME_TAB;
COMMIT;
```

If the value of bind option SQLRULES is STD, or if special register CURRENT RULES has been set in the program and has the value STD, DB2 creates the LOB table space, auxiliary table, and auxiliary index for you when you execute the ALTER statement to add the LOB column.

Now that your DB2 objects for the LOB data are defined, you can load your employee resumes into DB2. To do this in an SQL application, you can define a host variable to hold the resume, copy the resume data from a file into the host variable, and then execute an UPDATE statement to copy the data into DB2. Although the data goes into the auxiliary table, your UPDATE statement specifies the name of the base table. The C language declaration of the host variable might be:

```
SQL TYPE is CLOB (5K) resumedata;
```

The UPDATE statement looks like this:

```
UPDATE EMP SET EMP_RESUME=:resumedata
  WHERE EMPNO=:employeenum;
```

In this example, employeenum is a host variable that identifies the employee who is associated with a resume.

After your LOB data is in DB2, you can write SQL applications to manipulate the data. You can use most SQL statements with LOBs. For example, you can use statements like these to extract information about an employee's department from the resume:

```
EXEC SQL BEGIN DECLARE SECTION;
  long deptInfoBeginLoc;
  long deptInfoEndLoc;
  SQL TYPE IS CLOB_LOCATOR resume;
  SQL TYPE IS CLOB_LOCATOR deptBuffer;
EXEC SQL END DECLARE SECTION;
:
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT EMPNO, EMP_RESUME FROM EMP;
:
EXEC SQL FETCH C1 INTO :employeenum, :resume;
:
EXEC SQL SET  :deptInfoBeginLoc =
  POSSTR(:resumedata, 'Department Information');

EXEC SQL SET :deptInfoEndLoc =
  POSSTR(:resumedata, 'Education');

EXEC SQL SET :deptBuffer =
  SUBSTR(:resume, :deptInfoBeginLoc,
  :deptInfoEndLoc - :deptInfoBeginLoc);
```

These statements use host variables of data type large object locator (LOB locator). LOB locators let you manipulate LOB data without moving the LOB data into host variables. By using LOB locators, you need much smaller amounts of memory for your programs. LOB locators are discussed in "Using LOB locators to save storage" on page 236.

Sample LOB applications: Table 22 lists the sample programs that DB2 provides to assist you in writing applications to manipulate LOB data. All programs reside in data set DSN710.SDSNSAMP.

Table 22. LOB samples shipped with DB2

Member that contains source code	Language	Function
DSNTEJ7	JCL	Demonstrates how to create a table with LOB columns, an auxiliary table, and an auxiliary index. Also demonstrates how to load LOB data that is 32KB or less into a LOB table space.
DSN8DLPL	C	Demonstrates the use of LOB locators and UPDATE statements to move binary data into a column of type BLOB.
DSN8DLRV	C	Demonstrates how to use a locator to manipulate data of type CLOB.
DSNTEP2	PL/I	Demonstrates how to allocate an SQLDA for rows that include LOB data and use that SQLDA to describe an input statement and fetch data from LOB columns.

For instructions on how to prepare and run the sample LOB applications, see Part 2 of *DB2 Installation Guide*.

Declaring LOB host variables and LOB locators

When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator variables to point to the LOB data. See "Using LOB locators to save storage" on page 236 for information on what LOB locators are and when you should use them instead of host variables.

You can declare LOB host variables and LOB locators in assembler, C, C++, COBOL, FORTRAN, and PL/I. For each host variable or locator of SQL type BLOB, CLOB, or DBCLOB that you declare, DB2 generates an equivalent declaration that uses host language data types. When you refer to a LOB host variable or locator in an SQL statement, you must use the variable you specified in the SQL type declaration. When you refer to the host variable in a host language statement, you must use the variable that DB2 generates. See "Part 2. Coding SQL in your host application program" on page 61 for the syntax of LOB declarations in each language and for host language equivalents for each LOB type.

DB2 supports host variable declarations for LOBs with lengths of up to 2 GB - 1. However, the size of a LOB host variable is limited by the restrictions of the host language and the amount of storage available to the program.

The following examples show you how to declare LOB host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that DB2 generates.

Declarations of LOB host variables in assembler: Table 23 shows assembler language declarations for some typical LOB types.

Table 23. Example of assembler LOB variable declarations

You Declare this Variable	DB2 Generates this Variable
blob_var SQL TYPE IS BLOB 1M	blob_var DS 0FL4 blob_var_length DS FL4 blob_var_data DS CL65535[1] ORG blob_var_data+(1048476-65535)
clob_var SQL TYPE IS CLOB 40000K	clob_var DS 0FL4 clob_var_length DS FL4 clob_var_data DS CL65535[1] ORG clob_var_data +(40960000-65535)
dbclob-var SQL TYPE IS DBCLOB 4000K	dbclob_var DS 0FL4 dbclob_var_length DS FL4 dbclob_var_data DS GL65534[2] ORG dbclob_var_data+(8192000-65534)
blob_loc SQL TYPE IS BLOB_LOCATOR	blob_loc DS FL4
clob_loc SQL TYPE IS CLOB_LOCATOR	clob_loc DS FL4
dbclob_var SQL TYPE IS DBCLOB_LOCATOR	dbclob_loc DS FL4

Table 23. Example of assembler LOB variable declarations (continued)

You Declare this Variable	DB2 Generates this Variable

Notes:

1. Because assembler language allows character declarations of no more than 65535 bytes, DB2 separates the host language declarations for BLOB and CLOB host variables that are longer than 65535 bytes into two parts.

2. Because assembler language allows graphic declarations of no more than 65534 bytes, DB2 separates the host language declarations for DBCLOB host variables that are longer than 65534 bytes into two parts.

Declarations of LOB host variables in C: Table 24 shows C and C⁺⁺ language declarations for some typical LOB types.

Table 24. Examples of C language variable declarations

You Declare this Variable	DB2 Generates this Variable

SQL TYPE IS BLOB (1M) blob_var;	struct {
	     unsigned long length;
	     char data[1048576];
	 } blob_var;

SQL TYPE IS CLOB(40000K) clob_var;	struct {
	     unsigned long length;
	     char data[40960000];
	 } clob_var;

SQL TYPE IS DBCLOB (4000K) dbclob_var;	struct {
	     unsigned long length;
	     wchar_t data[4096000];
	 } dbclob_var;

SQL TYPE IS BLOB_LOCATOR blob_loc;	unsigned long blob_loc;
SQL TYPE IS CLOB_LOCATOR clob_loc;	unsigned long clob_loc;
SQL TYPE IS DBCLOB_LOCATOR dbclob_loc;	unsigned long dbclob_loc;

Declarations of LOB host variables in COBOL: Table 25 shows COBOL declarations for some typical LOB types.

Table 25. Examples of COBOL variable declarations

You Declare this Variable	DB2 Generates this Variable

01   BLOB-VAR USAGE IS
       SQL TYPE IS BLOB(1M).
	01 BLOB-VAR.
	     02 BLOB-VAR-LENGTH
	           PIC 9(9) COMP.
	     02 BLOB-VAR-DATA.
	         49 FILLER PIC X(32767).[1]
	         49 FILLER PIC X(32767).
	         Repeat 30 times
	     :
	     :
	         49 FILLER
	               PIC X(1048576-32*32767).

Table 25. Examples of COBOL variable declarations (continued)

You Declare this Variable	DB2 Generates this Variable

01 CLOB-VAR USAGE IS
    SQL TYPE IS CLOB(40000K).
	01 CLOB-VAR.
	    02 CLOB-VAR-LENGTH
	         PIC 9(9) COMP.
	    02 CLOB-VAR-DATA.
	         49 FILLER PIC X(32767).[1]
	         49 FILLER PIC X(32767).
	         Repeat 1248 times
	
	    ⋮
	
	         49 FILLER
	              PIC X(40960000-1250*32767).

01 DBCLOB-VAR USAGE IS
   SQL TYPE IS DBCLOB(4000K).
	01 DBCLOB-VAR.
	    02 DBCLOB-VAR-LENGTH
	         PIC 9(9) COMP.
	    02 DBCLOB-VAR-DATA.
	         49 FILLER PIC G(32767)
	              USAGE DISPLAY-1.[2]
	         49 FILLER PIC G(32767)
	              USAGE DISPLAY-1.
	         Repeat 1248 times
	
	    ⋮
	
	         49 FILLER
	              PIC X(20480000-1250*32767)
	              USAGE DISPLAY-1.

01 BLOB-LOC USAGE IS SQL TYPE IS BLOB-LOCATOR.	01 BLOB-LOC PIC S9(9) USAGE IS BINARY.
01 CLOB-LOC USAGE IS SQL TYPE IS CLOB-LOCATOR.	01 CLOB-LOC PIC S9(9) USAGE IS BINARY.
01 DBCLOB-LOC USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	01 DBCLOB-LOC PIC S9(9) USAGE IS BINARY.

Notes:

1. Because the COBOL language allows character declarations of no more than 32767 bytes, for BLOB or CLOB host variables that are greater than 32767 bytes in length, DB2 creates multiple host language declarations of 32767 or fewer bytes.

2. Because the COBOL language allows graphic declarations of no more than 32767 double-byte characters, for DBCLOB host variables that are greater than 32767 double-byte characters in length, DB2 creates multiple host language declarations of 32767 or fewer double-byte characters.

Declarations of LOB host variables in FORTRAN: Table 26 shows FORTRAN declarations for some typical LOB types.

Table 26. Examples of FORTRAN variable declarations

You Declare this Variable	DB2 Generates this Variable
SQL TYPE IS BLOB(1M) blob_var	CHARACTER blob_var(1048580) INTEGER*4 blob_var_LENGTH CHARACTER blob_var_DATA EQUIVALENCE(blob_var(1), + blob_var_LENGTH) EQUIVALENCE(blob_var(5), + blob_var_DATA)

Table 26. Examples of FORTRAN variable declarations (continued)

You Declare this Variable	DB2 Generates this Variable
SQL TYPE IS CLOB(40000K) clob_var	CHARACTER clob_var(4096004) INTEGER*4 clob_var_length CHARACTER clob_var_data EQUIVALENCE(clob_var(1), + clob_var_length) EQUIVALENCE(clob_var(5), + clob_var_data)
SQL TYPE IS BLOB_LOCATOR blob_loc	INTEGER*4 blob_loc
SQL TYPE IS CLOB_LOCATOR clob_loc	INTEGER*4 clob_loc

Declarations of LOB host variables in PL/I: Table 27 shows PL/I declarations for some typical LOB types.

Table 27. Examples of PL/I variable declarations

You Declare this Variable	DB2 Generates this Variable

DCL BLOB_VAR
   SQL TYPE IS BLOB (1M);
	DCL 1 BLOB_VAR,
	   2 BLOB_VAR_LENGTH FIXED BINARY(31),
	   2 BLOB_VAR_DATA,[1]
	      3 BLOB_VAR_DATA1(32)
	          CHARACTER(32767),
	      3 BLOB_VAR_DATA2
	          CHARACTER(1048576-32*32767);

DCL CLOB_VAR
   SQL TYPE IS CLOB (40000K);
	DCL 1 CLOB_VAR,
	   2 CLOB_VAR_LENGTH FIXED BINARY(31),
	   2 CLOB_VAR_DATA,[1]
	      3 CLOB_VAR_DATA1(1250)
	          CHARACTER(32767),
	      3 CLOB_VAR_DATA2
	          CHARACTER(40960000-1250*32767);

DCL DBCLOB_VAR
   SQL TYPE IS DBCLOB (4000K);
	DCL 1 DBCLOB_VAR,
	   2 DBCLOB_VAR_LENGTH FIXED BINARY(31),
	   2 DBCLOB_VAR_DATA,[2]
	      3 DBCLOB_VAR_DATA1(2500)
	          GRAPHIC(16383),
	      3 DBCLOB_VAR_DATA2
	          GRAPHIC(40960000-2500*16383);

DCL blob_loc SQL TYPE IS BLOB_LOCATOR;	DCL blob_loc FIXED BINARY(31);
DCL clob_loc SQL TYPE IS CLOB_LOCATOR;	DCL clob_loc FIXED BINARY(31);
DCL dbclob_loc SQL TYPE IS DBCLOB_LOCATOR;	DCL dbclob_loc FIXED BINARY(31);

Table 27. Examples of PL/I variable declarations (continued)

You Declare this Variable	DB2 Generates this Variable

Notes:

1. Because the PL/I language allows character declarations of no more than 32767 bytes, for BLOB or CLOB host variables that are greater than 32767 bytes in length, DB2 creates host language declarations in the following way:

    *   If the length of the LOB is greater than 32767 bytes and evenly divisible by 32767, DB2 creates an array of 32767-byte strings. The dimension of the array is *length*/32767.
    *   If the length of the LOB is greater than 32767 bytes but not evenly divisible by 32767, DB2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, *n*, is *length*/32767. The second is a character string of length *length-n**32767.

2. Because the PL/I language allows graphic declarations of no more than 16383 double-byte characters, DB2 creates host language declarations in the following way:

    *   If the length of the LOB is greater than 16383 characters and evenly divisible by 16383, DB2 creates an array of 16383-character strings. The dimension of the array is *length*/16383.
    *   If the length of the LOB is greater than 16383 characters but not evenly divisible by 16383, DB2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, *m*, is *length*/16383. The second is a character string of length *length-m**16383.

LOB materialization

LOB materialization means that DB2 places a LOB value value into contiguous storage in a data space. Because LOB values can be very large, DB2 avoids materializing LOB data until absolutely necessary. However, DB2 must materialize LOBs when your application program:
*   Calls a user-defined function with a LOB as an argument
*   Moves a LOB into or out of a stored procedure
*   Assigns a LOB host variable to a LOB locator host variable
*   Converts a LOB from one CCSID to another

Data spaces for LOB materialization: The amount of storage that is used in data spaces for LOB materialization depends on a number of factors including:
*   The size of the LOBs
*   The number of LOBs that need to be materialized in a statement

DB2 allocates a certain number of data spaces for LOB materialization. If there is insufficient space available in a data space for LOB materialization, your application receives SQLCODE -904.

Although you cannot completely avoid LOB materialization, you can minimize it by using LOB locators, rather than LOB host variables in your application programs. See "Using LOB locators to save storage" for information on how to use LOB locators.

Using LOB locators to save storage

To retrieve LOB data from a DB2 table, you can define host variables that are large enough to hold all of the LOB data. This requires your application to allocate large amounts of storage, and requires DB2 to move large amounts of data, which can be inefficient or impractical. Instead, you can use LOB locators. LOB locators let

you manipulate LOB data without retrieving the data from the DB2 table. Using LOB locators for LOB data retrieval is a good choice in the following situations:

- When you move only a small part of a LOB to a client program
- When the entire LOB does not fit in the application's memory
- When the program needs a temporary LOB value from a LOB expression but does not need to save the result
- When performance is important

A LOB locator is associated with a LOB value or expression, not with a row in a DB2 table or a physical storage location in a table space. Therefore, after you select a LOB value using a locator, the value in the locator normally does not change until the current unit of work ends. However the value of the LOB itself can change.

If you want to remove the association between a LOB locator and its value before a unit of work ends, execute the FREE LOCATOR statement. To keep the association between a LOB locator and its value after the unit of work ends, execute the HOLD LOCATOR statement. After you execute a HOLD LOCATOR statement, the locator keeps the association with the corresponding value until you execute a FREE LOCATOR statement or the program ends.

If you execute HOLD LOCATOR or FREE LOCATOR dynamically, you cannot use EXECUTE IMMEDIATE. For more information on HOLD LOCATOR and FREE LOCATOR, see Chapter 5 of *DB2 SQL Reference*.

Deferring evaluation of a LOB expression to improve performance

DB2 moves no bytes of a LOB value until a program assigns a LOB expression to a target destination. This means that when you use a LOB locator with string functions and operators, you can create an expression that DB2 does not evaluate until the time of assignment. This is called *deferring evaluation* of a LOB expression. Deferring evaluation can improve LOB I/O performance.

The following example is a C language program that defers evaluation of a LOB expression. The program runs on a client and modifies LOB data at a server. The program searches for a particular resume (EMPNO = '000130') in the EMP_RESUME table. It then uses LOB locators to rearrange a copy of the resume (with EMPNO = 'A00130'). In the copy, the Department Information Section appears at the end of the resume. The program then inserts the copy into EMP_RESUME without modifying the original resume.

Because the program uses LOB locators, rather than placing the LOB data into host variables, no LOB data is moved until the INSERT statement executes. In addition, no LOB data moves between the client and the server.

```
EXEC SQL INCLUDE SQLCA;

/**************************/
/* Declare host variables */                                    1
/**************************/
EXEC SQL BEGIN DECLARE SECTION;
   char userid[9];
   char passwd[19];
   long            HV_START_DEPTINFO;
   long            HV_START_EDUC;
   long            HV_RETURN_CODE;
   SQL TYPE IS CLOB_LOCATOR HV_NEW_SECTION_LOCATOR;
   SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR1;
   SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR2;
   SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR3;
EXEC SQL END DECLARE SECTION;
```

Figure 95. Example of deferring evaluation of LOB expressions (Part 1 of 2)

```
/**************************************************/
/* Delete any instance of "000130" from previous */
/* executions of this sample                      */
/**************************************************/
EXEC SQL DELETE FROM EMP_RESUME WHERE EMPNO = 'A00130';

/**************************************************/
/* Use a single row select to get the document   */       2
/**************************************************/
EXEC SQL SELECT RESUME
           INTO :HV_DOC_LOCATOR1
           FROM EMP_RESUME
          WHERE EMPNO = '000130'
            AND RESUME_FORMAT = 'ascii';
/****************************************************/
/* Use the POSSTR function to locate the start of   */
/* sections "Department Information" and "Education" */       3
/****************************************************/
EXEC SQL SET :HV_START_DEPTINFO =
          POSSTR(:HV_DOC_LOCATOR1, 'Department Information');

EXEC SQL SET :HV_START_EDUC =
            POSSTR(:HV_DOC_LOCATOR1, 'Education');

/*****************************************************/
/* Replace Department Information section with nothing */
/*****************************************************/
EXEC SQL SET :HV_DOC_LOCATOR2 =
     SUBSTR(:HV_DOC_LOCATOR1, 1, :HV_START_DEPTINFO -1)
  || SUBSTR (:HV_DOC_LOCATOR1, :HV_START_EDUC);
/*****************************************************/
/* Associate a new locator with the Department       */
/* Information section                               */
/*****************************************************/
EXEC SQL SET :HV_NEW_SECTION_LOCATOR =
     SUBSTR(:HV_DOC_LOCATOR1, :HV_START_DEPTINFO,
    :HV_START_EDUC -:HV_START_DEPTINFO);

/*****************************************************/
/* Append the Department Information to the end       */
/* of the resume                                     */
/*****************************************************/
EXEC SQL SET :HV_DOC_LOCATOR3 =
  :HV_DOC_LOCATOR2 || :HV_NEW_SECTION_LOCATOR;
/*****************************************************/
/* Store the modified resume in the table.  This is  */       4
/* where the LOB data really moves.                  */
/*****************************************************/
EXEC SQL INSERT INTO EMP_RESUME VALUES ('A00130', 'ascii',
   :HV_DOC_LOCATOR3, DEFAULT);

/********************/
/* Free the locators */                                        5
/********************/
EXEC SQL FREE LOCATOR :HV_DOC_LOCATOR1, :HV_DOC_LOCATOR2, :HV_DOC_LOCATOR3;
```

Figure 95. Example of deferring evaluation of LOB expressions (Part 2 of 2)

Notes on Figure 95 on page 238:

1   Declare the LOB locators here.

2   This SELECT statement associates LOB locator HV_DOC_LOCATOR1 with the value of column RESUME for employee number 000130.

3   The next five SQL statements use LOB locators to manipulate the resume data without moving the data.

|4| Evaluation of the LOB expressions in the previous statements has been deferred until execution of this INSERT statement.

|5| Free all LOB locators to release them from their associated values.

Indicator variables and LOB locators

For host variables other than LOB locators, when you select a null value into a host variable, DB2 assigns a negative value to the associated indicator variable. However, for LOB locators, DB2 uses indicator variables differently. A LOB locator is never null. When you select a LOB column using a LOB locator and the LOB column contains a null value, DB2 assigns a null value to the associated indicator variable. The value in the LOB locator does not change. In a client/server environment, this null information is recorded only at the client.

When you use LOB locators to retrieve data from columns that can contain null values, define indicator variables for the LOB locators, and check the indicator variables after you fetch data into the LOB locators. If an indicator variable is null after a fetch operation, you cannot use the value in the LOB locator.

Valid assignments for LOB locators

Although you usually use LOB locators for assigning data to and retrieving data from LOB columns, you can also use LOB locators to assign data to CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC columns. However, you cannot fetch data from CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC columns into LOB locators.

Chapter 14. Creating and using user-defined functions

A user-defined function is an extension to the SQL language. A user-defined function is similar to a host language subprogram or function. However, a user-defined function is often the better choice for an SQL application because you can invoke a user-defined function in an SQL statement.

This chapter presents the following information about user-defined functions:
- "Overview of user-defined function definition, implementation, and invocation"
- "Defining a user-defined function" on page 244
- "Implementing an external user-defined function" on page 248
- "Implementing an SQL scalar function" on page 287
- "Invoking a user-defined function" on page 288

This chapter contains information that applies to all user-defined functions and specific information about user-defined functions in languages other than Java™. For information on writing, preparing, and running Java user-defined functions, see *DB2 Application Programming Guide and Reference for Java*.

Overview of user-defined function definition, implementation, and invocation

The two types of user-defined functions are:
- *Sourced* user-defined functions, which are based on existing built-in functions or user-defined functions
- *External* user-defined functions, which a programmer writes in a host language
- *SQL* user-defined functions, which contain the source code for the user-defined function in the user-defined function definition.

User-defined functions can also be categorized as a *user-defined scalar functions* or a *user-defined table functions*:
- A user-defined scalar function returns a single-value answer each time it is invoked.
- A user-defined table function returns a table to the SQL statement that references it.

External user-defined functions can be user-defined scalar functions or user-defined table functions. Sourced and SQL user-defined functions cannot be user-defined table functions.

Creating and using a user-defined function involves these steps:
- Setting up the environment for user-defined functions

  A system administrator probably performs this step. The user-defined function environment is shown in Figure 96 on page 242. The steps for setting up and maintaining the user-defined function environment are the same as for setting up and maintaining the environment for stored procedures in WLM-established address spaces. See "Chapter 24. Using stored procedures for client/server processing" on page 525 for this information.
- Writing and preparing the user-defined function

  This step is necessary only for an external user-defined function.

  The person who performs this step is called the user-defined function *implementer*.

- Defining the user-defined function to DB2

  The person who performs this step is called the user-defined function *definer*.
- Invoking the user-defined function from an SQL application

  The person who performs this step is called the user-defined function *invoker*.
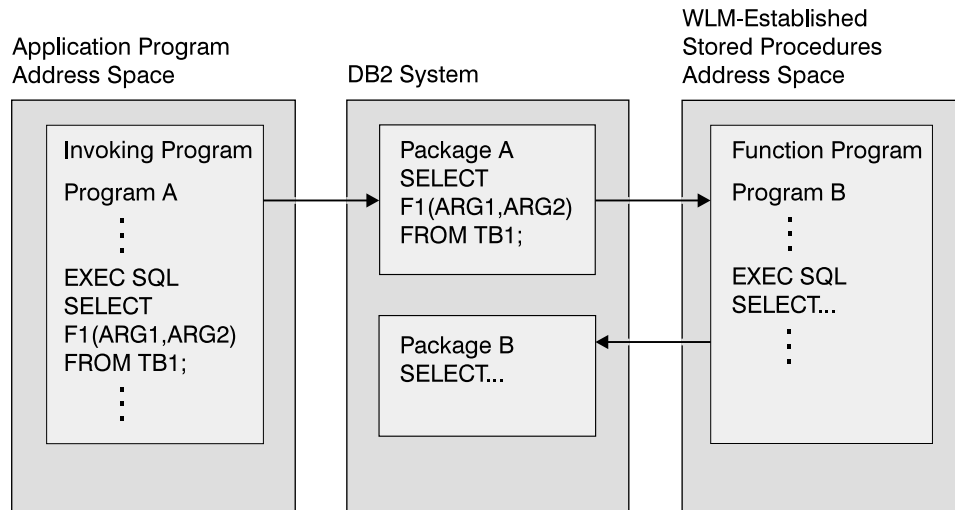
Figure 96. The user-defined function environment

Example of creating and using a user-defined scalar function

Suppose that your organization needs a user-defined scalar function that calculates the bonus that each employee receives. All employee data, including salaries, commissions, and bonuses, is kept in the employee table, EMP. The input fields for the bonus calculation function are the values of the SALARY and COMM columns. The output from the function goes into the BONUS column. Because this function gets its input from a DB2 table and puts the output in a DB2 table, a convenient way to manipulate the data is through a user-defined function.

The user-defined function's definer and invoker determine that this new user-defined function should have these characteristics:
- The user-defined function name is CALC_BONUS.
- The two input fields are of type DECIMAL(9,2).
- The output field is of type DECIMAL(9,2).
- The program for the user-defined function is written in COBOL and has a load module name of CBONUS.

Because no built-in function or user-defined function exists on which to build a sourced user-defined function, the function implementer must code an external user-defined function. The implementer performs the following steps:
- Writes the user-defined function, which is a COBOL program
- Precompiles, compiles, and links the program
- Binds a package if the user-defined function contains SQL statements
- Tests the program thoroughly
- Grants execute authority on the user-defined function package to the definer

The user-defined function definer executes this CREATE FUNCTION statement to register CALC_BONUS to DB2:

```
CREATE FUNCTION CALC_BONUS(DECIMAL(9,2),DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'CBONUS'
  PARAMETER STYLE DB2SQL
  LANGUAGE COBOL;
```

The definer then grants execute authority on CALC_BONUS to all invokers.

User-defined function invokers write and prepare application programs that invoke CALC_BONUS. An invoker might write a statement like this, which uses the user-defined function to update the BONUS field in the employee table:

```
UPDATE EMP
  SET BONUS = CALC_BONUS(SALARY,COMM);
```

An invoker can execute this statement either statically or dynamically.

User-defined function samples shipped with DB2

To assist you in defining, implementing, and invoking your user-defined functions, DB2 provides a number of sample user-defined functions. All user-defined function code is in data set DSN710.SDSNSAMP.

Table 28 summarizes the characteristics of the sample user-defined functions.

Table 28. User-defined function samples shipped with DB2

User-defined function name	Language	Member that contains source code	Purpose
ALTDATE[1]	C	DSN8DUAD	Converts the current date to a user-specified format
ALTDATE[2]	C	DSN8DUCD	Converts a date from one format to another
ALTTIME[3]	C	DSN8DUAT	Converts the current time to a user-specified format
ALTTIME[4]	C	DSN8DUCT	Converts a time from one format to another
DAYNAME	C++	DSN8EUDN	Returns the day of the week for a user-specified date
MONTHNAME	C++	DSN8EUMN	Returns the month for a user-specified date
CURRENCY	C	DSN8DUCY	Formats a floating-point number as a currency value
TABLE_NAME	C	DSN8DUTI	Returns the unqualified table name for a table, view, or alias
TABLE_QUALIF	C	DSN8DUTI	Returns the qualifier for a table, view, or alias
TABLE_LOCATION	C	DSN8DUTI	Returns the location for a table, view, or alias
WEATHER	C	DSN8DUWF	Returns a table of weather information from a EBCDIC data set

Table 28. User-defined function samples shipped with DB2 (continued)

User-defined function name	Language	Member that contains source code	Purpose

Notes:

1. This version of ALTDATE has one input parameter, of type VARCHAR(13).

2. This version of ALTDATE has three input parameters, of type VARCHAR(17), VARCHAR(13), and VARCHAR(13).

3. This version of ALTTIME has one input parameter, of type VARCHAR(14).

4. This version of ALTTIME has three input parameters, of type VARCHAR(11), VARCHAR(14), and VARCHAR(14).

Member DSN8DUWC contains a client program that shows you how to invoke the WEATHER user-defined table function.

Member DSNTEJ2U shows you how to define and prepare the sample user-defined functions and the client program.

Defining a user-defined function

Before you can define a user-defined function to DB2, you must determine the characteristics of the user-defined function, such as the user-defined function name, schema (qualifier), and number and data types of the input parameters and the types of the values returned. Then you execute a CREATE FUNCTION statement to register the information in the DB2 catalog. If you discover after you define the function that any of these characteristics is not appropriate for the function, you can use an ALTER FUNCTION statement to change information in the definition. You cannot use ALTER FUNCTION to change some of the characteristics of a user-defined function definition. See Chapter 5 of *DB2 SQL Reference* for information on which characteristics you can change with ALTER FUNCTION.

Components of a user-defined function definition

The characteristics you include in a CREATE FUNCTION or ALTER FUNCTION statement depend on whether the user-defined function is external or sourced. Table 29 lists the characteristics of a user-defined function, the corresponding parameters in the CREATE FUNCTION and ALTER FUNCTION statements, and which parameters are valid for sourced and external user-defined functions.

Table 29. Characteristics of a user-defined function

Characteristic	CREATE FUNCTION or ALTER FUNCTION parameter	Valid in sourced function?	Valid in external function?	Valid in SQL function?
User-defined function name	FUNCTION	Yes	Yes	Yes
Input parameter types and encoding schemes	FUNCTION	Yes	Yes	Yes
Output parameter types and encoding schemes	RETURNS RETURNS TABLE[1]	Yes	Yes	Yes[2]
Specific name	SPECIFIC	Yes	Yes	Yes
External name	EXTERNAL NAME	No	Yes	No

Table 29. Characteristics of a user-defined function (continued)

Characteristic	CREATE FUNCTION or ALTER FUNCTION parameter	Valid in sourced function?	Valid in external function?	Valid in SQL function?
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE PLI LANGUAGE JAVA LANGUAGE SQL	No	Yes	Yes[3]
Deterministic or not deterministic	NOT DETERMINISTIC DETERMINISTIC	No	Yes	Yes
Types of SQL statements in the function	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA	No	Yes[4]	Yes[5]
Name of source function	SOURCE	Yes	No	No
Parameter style	PARAMETER STYLE DB2SQL PARAMETER STYLE JAVA	No	Yes	Yes
Address space for user-defined functions	FENCED	No	Yes	Yes
Call with null input	RETURNS NULL ON NULL INPUT CALLED ON NULL INPUT	No	Yes	Yes
External actions	EXTERNAL ACTION NO EXTERNAL ACTION	No	Yes	Yes
Scratchpad specification	NO SCRATCHPAD SCRATCHPAD *length*	No	Yes	Yes
Call function after SQL processing	NO FINAL CALL FINAL CALL	No	Yes	Yes
Consider function for parallel processing	ALLOW PARALLEL DISALLOW PARALLEL	No	Yes[4]	Yes
Package collection	NO COLLID COLLID *collection-id*	No	Yes	No
WLM environment	WLM ENVIRONMENT *name* WLM ENVIRONMENT *name,**	No	Yes	No
CPU time for a function invocation	ASUTIME NO LIMIT ASUTIME LIMIT *integer*	No	Yes	Yes
Load module stays in memory	STAY RESIDENT NO STAY RESIDENT YES	No	Yes	Yes
Program type	PROGRAM TYPE MAIN PROGRAM TYPE SUB	No	Yes	Yes
Security	SECURITY DB2 SECURITY USER SECURITY DEFINER	No	Yes	Yes
Run-time options	RUN OPTIONS *options*	No	Yes	Yes
Pass DB2 environment information	NO DBINFO DBINFO	No	Yes	Yes
Expected number of rows returned	CARDINALITY *integer*	No	Yes[1]	No

Table 29. Characteristics of a user-defined function (continued)

Characteristic	CREATE FUNCTION or ALTER FUNCTION parameter	Valid in sourced function?	Valid in external function?	Valid in SQL function?
Function resolution is based on the declared parameter types	STATIC DISPATCH	No	No	Yes
SQL expression that evaluates to the value returned by the function	RETURN *expression*	No	No	Yes
Encoding scheme for all string parameters	PARAMETER CCSID EBCDIC PARAMETER CCSID ASCII PARAMETER CCSID UNICODE	No	Yes	Yes

Notes:

1. RETURNS TABLE and CARDINALITY are valid only for user-defined table functions.

2. An SQL user-defined function can return only one parameter.

3. LANGUAGE SQL is valid only for an SQL user-defined function.

4. MODIFIES SQL DATA and ALLOW PARALLEL are not valid for user-defined table functions.

5. MODIFIES SQL DATA and NO SQL are not valid for SQL user-defined functions.

For a complete explanation of the parameters in a CREATE FUNCTION or ALTER FUNCTION statement, see Chapter 5 of *DB2 SQL Reference*.

Examples of user-defined function definitions

Example: Definition for an external user-defined scalar function: A programmer has written a user-defined function that searches for a string of maximum length 200 in a CLOB value whose maximum length is 500 KB. The output from the user-defined function is of type float, but users require integer output for their SQL statements. The user-defined function is written in C and contains no SQL statements. This CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC FINDSTRINCLOB
  EXTERNAL NAME 'FINDSTR'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED;
```

Example: Definition for an external user-defined scalar function that overloads an operator: A programmer has written a user-defined function that overloads the built-in SQL division operator (/). That is, this user-defined function is invoked when an application program executes a statement like either of the following:

```
UPDATE TABLE1 SET INTCOL1=INTCOL2/INTCOL3;
```

```
UPDATE TABLE1 SET INTCOL1="/"(INTCOL2,INTCOL3);
```

The user-defined function takes two integer values as input. The output from the user-defined function is of type integer. The user-defined function is in the MATH schema, is written in assembler, and contains no SQL statements. This CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION MATH."/" (INT, INT)
  RETURNS INTEGER
  SPECIFIC DIVIDE
  EXTERNAL NAME 'DIVIDE'
  LANGUAGE ASSEMBLE
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED;
```

Suppose you want the FINDSTRING user-defined function to work on BLOB data types, as well as CLOB types. You can define another instance of the user-defined function that specifies a BLOB type as input:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC FINDSTRINBLOB
  EXTERNAL NAME 'FNDBLOB'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED;
```

Each instance of FINDSTRING uses a different application program to implement the user-defined function.

Example: Definition for a sourced user-defined function: Suppose you need a user-defined function that finds a string in a value with a distinct type of BOAT. BOAT is based on a BLOB data type. User-defined function FINDSTRING has already been defined. FINDSTRING takes a BLOB data type and performs the required function. You can therefore define a sourced user-defined function based on FINDSTRING to do the string search on values of type BOAT. This CREATE FUNCTION statement defines the sourced user-defined function:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INTEGER
  SPECIFIC FINDSTRINBOAT
  SOURCE SPECIFIC FINDSTRINBLOB;
```

Example: Definition for a user-defined table function: An application programmer has written a user-defined function that receives two values and returns a table. The two input values are:
- A character string of maximum length 30 that describes a subject
- A character string of maximum length 255 that contains text to search for

The user-defined function scans documents on the subject for the search string and returns a list of documents that match the search criteria, with an abstract for each document. The list is in the form of a two-column table. The first column is a character column of length 16 that contains document IDs. The second column is a varying-character column of maximum length 5000 that contains document abstracts.

The user-defined function is written in COBOL, uses SQL only to perform queries, always produces the same output for given input, and should not execute as a parallel task. The program is reentrant, and successive invocations of the user-defined function share information. You expect an invocation of the user-defined function to return about 20 rows.

The following CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16), DOC_ABSTRACT VARCHAR(5000))
  EXTERNAL NAME 'DOCMTCH'
  LANGUAGE COBOL
  PARAMETER STYLE DB2SQL
  READS SQL DATA
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20;
```

Implementing an external user-defined function

This section discusses these steps in implementing an external user-defined function:
- "Writing a user-defined function"
- "Preparing a user-defined function for execution" on page 283
- "Testing a user-defined function" on page 285

Writing a user-defined function

A user-defined function is similar to any other SQL program. When you write a user-defined function, you can include static or dynamic SQL statements, IFI calls, and DB2 commands issued through IFI calls.

Your user-defined function can also access remote data using the following methods:
- DB2 private protocol access using three-part names or aliases for three-part names
- DRDA access using three-part names or aliases for three-part names
- DRDA access using CONNECT or SET CONNECTION statements

The user-defined function and the application that calls it can access the same remote site if both use the same protocol.

You can write an external user-defined function in assembler, C, C++, COBOL, PL/I, or Java. User-defined functions that are written in COBOL can include object-oriented extensions, just as other DB2 COBOL programs can. For information on writing Java user-defined functions, see *DB2 Application Programming Guide and Reference for Java*.

The following sections include additional information that you need when you write a user-defined function:
- "Restrictions on user-defined function programs" on page 249
- "Coding your user-defined function as a main program or as a subprogram" on page 249
- "Parallelism considerations" on page 249
- "Passing parameter values to and from a user-defined function" on page 251
- "Examples of passing parameters in a user-defined function" on page 264
- "Using special registers in a user-defined function" on page 275
- "Using a scratchpad in a user-defined function" on page 276
- "Accessing transition tables in a user-defined function or stored procedure" on page 278

Restrictions on user-defined function programs

Observe these restrictions when you write a user-defined function:

- Because DB2 uses the Recoverable Resource Manager Services attachment facility (RRSAF) as its interface with your user-defined function, you must not include RRSAF calls in your user-defined function. DB2 rejects any RRSAF calls that it finds in a user-defined function.

- If your user-defined function is not defined with parameters SCRATCHPAD or EXTERNAL ACTION, the user-defined function is not guaranteed to execute under the same task each time it is invoked.

- You cannot execute COMMIT or ROLLBACK statements in your user-defined function.

- You must close all open cursors in a user-defined scalar function. DB2 returns an SQL error if a user-defined scalar function does not close all cursors before it completes.

- When you choose the language in which to write a user-defined function program, be aware of restrictions on the number of parameters that can be passed to a routine in that language. User-defined table functions in particular can require large numbers of parameters. Consult the programming guide for the language in which you plan to write the user-defined function for information on the number of parameters that can be passed.

Coding your user-defined function as a main program or as a subprogram

You can code your user-defined function as either a main program or a subprogram. The way that you code your program must agree with the way you defined the user-defined function: with the PROGRAM TYPE MAIN or PROGRAM TYPE SUB parameter. The main difference is that when a main program starts, Language Environment allocates the application program storage that the external user-defined function uses. When a main program ends, Language Environment® closes files and releases dynamically allocated storage.

If you code your user-defined function as a subprogram and manage the storage and files yourself, you can get better performance. The user-defined function should always free any allocated storage before it exits. To keep data between invocations of the user-defined function, use a scratchpad.

You must code a user-defined table function that accesses external resources as a subprogram. Also ensure that the definer specifies the EXTERNAL ACTION parameter in the CREATE FUNCTION or ALTER FUNCTION statement. Program variables for a subprogram persist between invocations of the user-defined function, and use of the EXTERNAL ACTION parameter ensures that the user-defined function stays in the same address space from one invocation to another.

Parallelism considerations

If the definer specifies the parameter ALLOW PARALLEL in the definition of a user-defined scalar function, and the invoking SQL statement runs in parallel, the function can run under a parallel task. DB2 executes a separate instance of the user-defined function for each parallel task. When you write your function program, you need to understand how the following parameter values interact with ALLOW PARALLEL so that you can avoid unexpected results:

- SCRATCHPAD

  When an SQL statement invokes a user-defined function that is defined with the ALLOW PARALLEL parameter, DB2 allocates one scratchpad for each parallel task of each reference to the function. This can lead to unpredictable or incorrect results.

For example, suppose that the user-defined function uses the scratchpad to count the number of times it is invoked. If a scratchpad is allocated for each parallel task, this count is the number of invocations done by the *parallel task* and not for the entire SQL statement, which is not the desired result.

- FINAL CALL

If a user-defined function performs an external action, such as sending a note, for each final call to the function, one note is sent for each parallel task instead of once for the function invocation.

- EXTERNAL ACTION

Some user-defined functions with external actions can receive incorrect results if the function is executed by parallel tasks.

For example, if the function sends a note for each initial call to the function, one note is sent for each parallel task instead of once for the function invocation.

- NOT DETERMINISTIC

A user-defined function that is not deterministic can generate incorrect results if it is run under a parallel task.

For example, suppose that you execute the following query under parallel tasks:

```
SELECT * FROM T1 WHERE C1 = COUNTER();
```

COUNTER is a user-defined function that increments a variable in the scratchpad every time it is invoked. Counter is nondeterministic because the same input does not always produce the same output. Table T1 contains one column, C1, that has these values:

```
1
2
3
4
5
6
7
8
9
10
```

When the query is executed with no parallelism, DB2 invokes COUNTER once for each row of table T1, and there is one scratchpad for counter, which DB2 initializes the first time that COUNTER executes. COUNTER returns 1 the first time it executes, 2 the second time, and so on. The result table for the query is therefore:

```
1
2
3
4
5
6
7
8
9
10
```

Now suppose that the query is run with parallelism, and DB2 creates three parallel tasks. DB2 executes the predicate WHERE C1 = COUNTER() for each parallel task. This means that each parallel task invokes its own instance of the user-defined function and has its own scratchpad. DB2 initializes the scratchpad to zero on the first call to the user-defined function for each parallel task.

If parallel task 1 processes rows 1 to 3, parallel task 2 processes rows 4 to 6, and parallel task 3 processes rows 7 to 10, the following results occur:

- When parallel task 1 executes, C1 has values 1, 2, and 3, and COUNTER returns values 1, 2, and 3, so the query returns values 1, 2, and 3.
- When parallel task 2 executes, C1 has values 4, 5, and 6, but COUNTER returns values 1, 2, and 3, so the query returns no rows.
- When parallel task 3, executes, C1 has values 7, 8, 9, and 10, but COUNTER returns values 1, 2, 3, and 4, so the query returns no rows.

Thus, instead of returning the 10 rows that you might expect from the query, DB2 returns only 3 rows.

Passing parameter values to and from a user-defined function

To receive parameters from and pass parameters to a function invoker, you must understand the structure of the parameter list, the meaning of each parameter, and whether DB2 or your user-defined function sets the value of each parameter. This section explains the parameters and gives examples of how a user-defined function in each host language receives the parameter list.

Figure 97 on page 252 shows the structure of the parameter list that DB2 passes to a user-defined function. An explanation of each parameter follows.

Register 1 ─────► Addresses of: Data:

Addresses of:		Data:
Parameter 1	───────►	Parameter 1 data
Parameter 2	───────►	Parameter 2 data
Result 1 [1]	───────►	Result 1 data
Result 2	───────►	Result 2 data
Indicator 1	───────►	Indicator 1
Indicator 2	───────►	Indicator 2
Result indicator 1 [1]	───────►	Result indicator 1
Result indicator 2	───────►	Result indicator 2
SQLSTATE	───────►	SQLSTATE
Procedure name	───────►	Procedure name
Specific name	───────►	Specific name
Message text	───────►	Message text
Scratchpad [2]	·············▷	Scratchpad
Call type [3]	·············▷	Call type
DBINFO [4, 5]	·············▷	DBINFO

[1] For a user-defined scalar function, only one result and one result indicator are passed.

[2] Passed if the SCRATCHPAD option is specified in the user-defined function definition.

[3] Passed if the FINAL CALL option is specified in a user-defined scalar function definition; always passed for a user-defined table function.

[4] For PL/I, this value is the address of a pointer to the DBINFO data..

[5] Passed if the DBINFO option is specified in the user-defined function definition

Figure 97. Parameter conventions for a user-defined function

Input parameter values: DB2 obtains the input parameters from the invoker's parameter list, and your user-defined function receives those parameters according to the rules of the host language in which the user-defined function is written. The number of input parameters is the same as the number of parameters in the user-defined function invocation. If one of the parameters in the function invocation is an expression, DB2 evaluates the expression and assigns the result of the expression to the parameter.

For all data types except LOBs, ROWIDs, and locators, see the tables listed in Table 30 for the host data types that are compatible with the data types in the user-defined function definition. For LOBs, ROWIDs, and locators, see tables Table 31, Table 32, Table 33 on page 254, and Table 34 on page 255.

Table 30. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	Table 8 on page 115
C	Table 10 on page 133
COBOL	Table 13 on page 156
PL/I	Table 17 on page 184

Table 31. Compatible assembler language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	Assembler declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	DS FL4
BLOB(*n*)	If n <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CLn If n > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(*n*-65535)
CLOB(*n*)	If n <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CLn If n > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(*n*-65535)
DBCLOB(*n*)	If m (=2*n) <= 65534: var DS 0FL4 var_length DS FL4 var_data DS CLm If m > 65534: var DS 0FL4 var_length DS FL4 var_data DS CL65534 ORG var_data+(*m*-65534)
ROWID	DS HL2,CL40

Table 32. Compatible C language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	C declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	unsigned long

Table 32. Compatible C language declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	C declaration
BLOB(*n*)	struct {unsigned long length; char data[n]; } var;
CLOB(*n*)	struct {unsigned long length; char var_data[n]; } var;
DBCLOB(*n*)	struct {unsigned long length; wchar_t[*] data[n]; } var;

ROWID	struct {
	    short int length;
	    char data[40];
	} var;

Note: *The SQLUDF file, which is in data set DSN710.SDSNC.H, includes the typedef sqldbchar, which you can use instead of wchar_t. Using sqldbchar lets you manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in DB2. sqldbchar also makes applications easier to port to other DB2 platforms.

Table 33. Compatible COBOL declarations for LOBs, ROWIDs, and locators

SQL data type in definition	COBOL declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	01 var PIC S9(9) USAGE IS BINARY.

BLOB(*n*)	If n <= 32767:
	01 var.
	      49 var-LENGTH PIC 9(9)
	              USAGE COMP.
	      49 var-DATA PIC X(*n*).
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	              USAGE COMP.
	  02 var-DATA.
	      49 FILLER
	          PIC X(32767).
	      49 FILLER
	          PIC X(32767).
	.
	.
	      49 FILLER
	          PIC X(mod(*n*,32767)).

Table 33. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration

CLOB(*n*)	If n <= 32767:
	01 var.
	    49 var-LENGTH PIC 9(9)
	            USAGE COMP.
	    49 var-DATA PIC X(*n*).
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	            USAGE COMP.
	  02 var-DATA.
	      49 FILLER
	          PIC X(32767).
	      49 FILLER
	          PIC X(32767).
	    :
	    :
	      49 FILLER
	          PIC X(mod(*n*,32767)).

DBCLOB(*n*)	If n <= 32767:
	01 var.
	    49 var-LENGTH PIC 9(9)
	            USAGE COMP.
	    49 var-DATA PIC G(*n*)
	       USAGE DISPLAY-1.
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	            USAGE COMP.
	  02 var-DATA.
	      49 FILLER
	          PIC G(32767)
	       USAGE DISPLAY-1.
	      49 FILLER
	          PIC G(32767).
	       USAGE DISPLAY-1.
	    :
	    :
	      49 FILLER
	          PIC G(mod(*n*,32767))
	          USAGE DISPLAY-1.

ROWID	01 var.
	    49 var-LEN PIC 9(4)
	            USAGE COMP.
	    49 var-DATA PIC X(40).

Table 34. Compatible PL/I declarations for LOBs, ROWIDs, and locators

SQL data type in definition	PL/I
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	BIN FIXED(31)

Table 34. Compatible PL/I declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	PL/I

BLOB(*n*)	If n <= 32767:
	01 var,
	    03 var_LENGTH
	       BIN FIXED(31),
	    03 var_DATA
	       CHAR(*n*);
	If n > 32767:
	01 var,
	  02 var_LENGTH
	    BIN FIXED(31),
	  02 var_DATA,
	    03 var_DATA1(*n*)
	      CHAR(32767),
	    03 var_DATA2
	      CHAR(mod(*n*,32767));

CLOB(*n*)	If n <= 32767:
	01 var,
	    03 var_LENGTH
	       BIN FIXED(31),
	    03 var_DATA
	       CHAR(*n*);
	If n > 32767:
	01 var,
	  02 var_LENGTH
	    BIN FIXED(31),
	  02 var_DATA,
	    03 var_DATA1(*n*)
	      CHAR(32767),
	    03 var_DATA2
	      CHAR(mod(*n*,32767));

DBCLOB(*n*)	If n <= 16383:
	01 var,
	    03 var_LENGTH
	       BIN FIXED(31),
	    03 var_DATA
	       GRAPHIC(*n*);
	If n > 16383:
	01 var,
	  02 var_LENGTH
	    BIN FIXED(31),
	  02 var_DATA,
	    03 var_DATA1(*n*)
	      GRAPHIC(16383),
	    03 var_DATA2
	      GRAPHIC(mod(*n*,16383));

ROWID	CHAR(40) VAR;

Result parameters: Set these values in your user-defined function before exiting.
For a user-defined scalar function, you return one result parameter. For a
user-defined table function, you return the same number of parameters as columns
in the RETURNS TABLE clause of the CREATE FUNCTION statement. DB2
allocates a buffer for each result parameter value and passes the buffer address to
the user-defined function. Your user-defined function places each result parameter
value in its buffer. You must ensure that the length of the value you place in each

output buffer does not exceed the buffer length. Use the SQL data type and length in the CREATE FUNCTION statement to determine the buffer length.

See "Passing parameter values to and from a user-defined function" on page 251 to determine the host data type to use for each result parameter value. If the CREATE FUNCTION statement contains a CAST FROM clause, use a data type that corresponds to the SQL data type in the CAST FROM clause. Otherwise, use a data type that corresponds to the SQL data type in the RETURNS or RETURNS TABLE clause.

To improve performance for user-defined table functions that return many columns, you can pass values for a subset of columns to the invoker. For example, a user-defined table function might be defined to return 100 columns, but the invoker needs values for only two columns. Use the DBINFO parameter to indicate to DB2 the columns for which you will return values. Then return values for only those columns. See the explanation of DBINFO below for information on how to indicate the columns of interest.

Input parameter indicators: These are SMALLINT values, which DB2 sets before it passes control to the user-defined function. You use the indicators to determine whether the corresponding input parameters are null. The number and order of the indicators are the same as the number and order of the input parameters. On entry to the user-defined function, each indicator contains one of these values:

0   The input parameter value is not null.

negative   The input parameter value is null.

Code the user-defined function to check all indicators for null values unless the user-defined function is defined with RETURNS NULL ON NULL INPUT. A user-defined function defined with RETURNS NULL ON NULL INPUT executes only if all input parameters are not null.

Result indicators: These are SMALLINT values, which you must set before the user-defined function ends to indicate to the invoking program whether each result parameter value is null. A user-defined scalar function has one result indicator. A user-defined table function has the same number of result indicators as the number of result parameters. The order of the result indicators is the same as the order of the result parameters. Set each result indicator to one of these values:

0 or positive The result parameter is not null.

negative   The result parameter is null.

SQLSTATE value: This is a CHAR(5) value, which you must set before the user-defined function ends. The user-defined function can return one of these SQLSTATE values:

00000   Use this value to indicate that the user-defined function executed without any warnings or errors.

01H*xx*   Use these values to indicate that the user-defined function detected a warning condition. *xx* can be any two single-byte alphanumeric characters. DB2 returns SQLCODE +462 if the user-defined function sets the SQLSTATE to 01H*xx*.

02000   Use this value to indicate that there no more rows are to be returned from a user-defined table function.

38*yxx*   Use these values to indicate that the user-defined function detected

an error condition. *y* can be any single-byte alphanumeric character except 5. *xx* can be any two single-byte alphanumeric characters. However, if an SQL statement in the user-defined function returns one of the following SQLSTATEs, passing that SQLSTATE back to the invoker is recommended.

38001   The user-defined function attempted to execute an SQL statement, but the user-defined function is not defined with NO SQL. DB2 returns SQLCODE -487 with this SQLSTATE. The user-defined function attempted to execute an SQL statement, but the user-defined function is defined with NO SQL. DB2 returns SQLCODE -487 with this SQLSTATE.

38002   The user-defined function attempted to execute an SQL statement that requires that the user-defined function is defined with MODIFIES SQL DATA, but the user-defined function is not defined with MODIFIES SQL DATA. DB2 returns SQLCODE -577 with this SQLSTATE.

38003   The user-defined function executed a COMMIT or ROLLBACK statement, which are not permitted in a user-defined function. DB2 returns SQLCODE -751 with this SQLSTATE.

38004   The user-defined function attempted to execute an SQL statement that requires that the user-defined function is defined with READS SQL DATA or MODIFIES SQL DATA, but the user-defined function is not defined with either of these options. DB2 returns SQLCODE -579 with this SQLSTATE.

When your user-defined function returns an SQLSTATE of 38*yxx* other than one of the four listed above, DB2 returns SQLCODE -443.

If the user-defined function returns an SQLSTATE that is not permitted for a user-defined function, DB2 replaces that SQLSTATE with 39001 and returns SQLCODE -463.

If both the user-defined function and DB2 set an SQLSTATE value, DB2 returns its SQLSTATE value to the invoker.

User-defined function name: DB2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(137): 8 bytes for the schema name, 1 byte for a period, and 128 bytes for the user-defined function name. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.

Specific name: DB2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(128) and is either the specific name from the CREATE FUNCTION statement or a specific name that DB2 generated. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.

Diagnostic message: This is a VARCHAR(70) value, which your user-defined function can set before exiting. Use this area to pass descriptive information about an error or warning to the invoker.

DB2 allocates a 70-byte buffer for this area and passes you the buffer address in the parameter list. Ensure that you do not write more than 70 bytes to the buffer. At least the first 17 bytes of the value you put in the buffer appear in the SQLERRMC field of the SQLCA that is returned to the invoker. The exact number of bytes depends on the number of other tokens in SQLERRMC. Do not use X'FF' in your diagnostic message. DB2 uses this value to delimit tokens.

Scratchpad: If the definer specified SCRATCHPAD in the CREATE FUNCTION statement, DB2 allocates a buffer for the scratchpad area and passes its address to the user-defined function. Before the user-defined function is invoked for the first time in an SQL statement, DB2 sets the length of the scratchpad in the first 4 bytes of the buffer and then sets the scratchpad area to X'00'. DB2 does not reinitialize the scratchpad between invocations of a correlated subquery.

You must ensure that your user-defined function does not write more bytes to the scratchpad than the scratchpad length.

Call type: For a user-defined scalar function, if the definer specified FINAL CALL in the CREATE FUNCTION statement, DB2 passes this parameter to the user-defined function. For a user-defined table function, DB2 always passes this parameter to the user-defined function.

On entry to a *user-defined scalar function,* the call type parameter has one of the following values:

-1   This is the *first call* to the user-defined function for the SQL statement. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.

0   This is a *normal call*. For a normal call, all the input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

1   This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

      This type of final call occurs when the invoking application explicitly closes a cursor. When a value of 1 is passed to a user-defined function, the user-defined function can execute SQL statements.

255   This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

      This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because DB2 might have already closed cursors before the final call.

During the first call, your user-defined scalar function should acquire any system resources it needs. During the final call, the user-defined scalar function should release any resources it acquired during the first call. The user-defined scalar function should return a result value only during normal calls. DB2 ignores any results that are returned during a final call. However, the user-defined scalar function can set the SQLSTATE and diagnostic message area during the final call.

If an invoking SQL statement contains more than one user-defined scalar function, and one of those user-defined functions returns an error SQLSTATE, DB2 invokes all of the user-defined functions for a final call, and the invoking SQL statement receives the SQLSTATE of the first user-defined function with an error.

On entry to a *user-defined table function,* the call type parameter has one of the following values:

-2   This is the *first call* to the user-defined function for the SQL statement. A first call occurs only if the FINAL CALL keyword is specified in the user-defined function definition. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.

-1   This is the *open call* to the user-defined function by an SQL statement. If FINAL CALL is not specified in the user-defined function definition, all input parameters are passed to the user-defined function, and the scratchpad, if allocated, is set to binary zeros during the open call. If FINAL CALL is specified for the user-defined function, DB2 does not modify the scratchpad.

0   This is a *fetch call* to the user-defined function by an SQL statement. For a fetch call, all input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

1   This is a *close call*. For a close call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

2   This is a *final call*. This type of final call occurs only if FINAL CALL is specified in the user-defined function definition. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

    This type of final call occurs when the invoking application executes a CLOSE CURSOR statement.

255   This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

    This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because DB2 might have already closed cursors before the final call.

If a user-defined table function is defined with FINAL CALL, the user-defined function should allocate any resources it needs during the first call and release those resources during the final call that sets a value of 2.

If a user-defined table function is defined with NO FINAL CALL, the user-defined function should allocate any resources it needs during the open call and release those resources during the close call.

During a fetch call, the user-defined table function should return a row. If the user-defined function has no more rows to return, it should set the SQLSTATE to 02000.

During the close call, a user-defined table function can set the SQLSTATE and diagnostic message area.

If a user-defined table function is invoked from a subquery, the user-defined table function receives a CLOSE call for each invocation of the subquery within the higher level query, and a subsequent OPEN call for the next invocation of the subquery within the higher level query.

DBINFO: If the definer specified DBINFO in the CREATE FUNCTION statement, DB2 passes the DBINFO structure to the user-defined function. DBINFO contains information about the environment of the user-defined function caller. It contains the following fields, in the order shown:

Location name length
    An unsigned 2-byte integer field. It contains the length of the location name in the next field.

Location name
    A 128-byte character field. It contains the name of the location to which the invoker is currently connected.

Authorization ID length
    An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

Authorization ID
    A 128-byte character field. It contains the authorization ID of the application from which the user-defined function is invoked, padded on the right with blanks. If this user-defined function is nested within other user-defined functions, this value is the authorization ID of the application that invoked the highest-level user-defined function.

Subsystem code page
    A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs and encoding scheme of the subsystem from which the user-defined function is invoked. The first nine fields are arranged in an array of three inner structures, each of which contains three integer fields. The three fields in each inner structure contain an SBCS, a DBCS, and a mixed CCSID. The first of the three inner structures is for EBCDIC CCSIDs. The second inner structure is for ASCII CCSIDs. The third inner structure is for Unicode CCSIDs. The last integer field in the outer structure is an index into the array of inner structures.

Table qualifier length
    An unsigned 2-byte integer field. It contains the length of the table qualifier in the next field. If the table name field is not used, this field contains 0.

Table qualifier
    A 128-byte character field. It contains the qualifier of the table that is specified in the table name field.

Table name length
An unsigned 2-byte integer field. It contains the length of the table name in the next field. If the table name field is not used, this field contains 0.

Table name
A 128-byte character field. This field contains the name of the table that the UPDATE or INSERT modifies if the reference to the user-defined function in the invoking SQL statement is in one of the following places:
* The right side of a SET clause in an UPDATE statement
* In the VALUES list of an INSERT statement

Otherwise, this field is blank.

Column name length
An unsigned 2-byte integer field. It contains the length of the column name in the next field. If no column name is passed to the user-defined function, this field contains 0.

Column name
A 128-byte character field. This field contains the name of the column that the UPDATE or INSERT modifies if the reference to the user-defined function in the invoking SQL statement is in one of the following places:
* The right side of a SET clause in an UPDATE statement
* In the VALUES list of an INSERT statement

Otherwise, this field is blank.

Product information
An 8-byte character field that identifies the product on which the user-defined function executes. This field has the form *pppvvrrm*, where:
* *ppp* is a 3-byte product code:

  DSN   DB2 for OS/390 and z/OS

  ARI   DB2 Server for VSE & VM

  QSQ   DB2 for AS/400®

  SQL   DB2 Universal Database
* *vv* is a 2-digit version identifier.
* *rr* is a 2-digit release identifier.
* *m* is a 1-digit modification level identifier.

Operating system
A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function runs. The value is one of these:

  0   Unknown

  1   OS/2®

  3   Windows®

  4   AIX

  5   Windows NT®

  6   HP-UX

  7   Solaris

  8   OS/390

  13   Siemens Nixdorf

15   Windows 95

16   SCO Unix

Number of entries in table function column list
An unsigned 2-byte integer field.

Reserved area
24 bytes.

Table function column list pointer
If a table function is defined, this field is a pointer to an array that contains 1000 2-byte integers. DB2 dynamically allocates the array. If a table function is not defined, this pointer is null.

Only the first *n* entries, where *n* is the value in the field entitled number of entries in table function column list, are of interest. *n* is greater than or equal to 0 and less than or equal to the number result columns defined for the user-defined function in the RETURNS TABLE clause of the CREATE FUNCTION statement. The values correspond to the numbers of the columns that the invoking statement needs from the table function. A value of 1 means the first defined result column, 2 means the second defined result column, and so on. The values can be in any order. If *n* is equal to 0, the first array element is 0. This is the case for a statement like the following one, where the invoking statement needs no column values.

```
SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ
```

This array represents an opportunity for optimization. The user-defined function does not need to return all values for all the result columns of the table function. Instead, the user-defined function can return only those columns that are needed in the particular context, which you identify by number in the array. However, if this optimization complicates the user-defined function logic enough to cancel the perfomance benefit, you might choose to return every defined column.

Unique application identifier
This field is a pointer to a string that uniquely identifies the application's connection to DB2. The string is regenerated at for each connection to DB2.

The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a 1- to 8-character network ID, a period, and a 1- to 8-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

Reserved area
20 bytes.

See the following section for examples of declarations of passed parameters in each language. If you write your user-defined function in C or C++, you can use the declarations in member SQLUDF of DSN710.SDSNC.H for many of the passed parameters. To include SQLUDF, make these changes to your program:

- Put this statement in your source code:

  ```
  #include <sqludf.h>
  ```

- Include the DSN710.SDSNC.H data set in the SYSLIB concatenation for the compile step of your program preparation job.

- Specify the NOMARGINS and NOSEQUENCE options in the compile step of your program preparation job.

Examples of passing parameters in a user-defined function

The following examples show how a user-defined function that is written in each of the supported host languages receives the parameter list that is passed by DB2.

These examples assume that the user-defined function is defined with the SCRATCHPAD, FINAL CALL, and DBINFO parameters.

Assembler: Figure 98 shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For an assembler language user-defined function that is a subprogram, the conventions are the same. In either case, you must include the CEEENTRY and CEEEXIT macros.

```
MYMAIN   CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
         USING PROGAREA,R13

         L     R7,0(R1)           GET POINTER TO PARM1
         MVC   PARM1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF PARM1
         L     R7,4(R1)           GET POINTER TO PARM2
         MVC   PARM1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF PARM2
         L     R7,12(R1)          GET POINTER TO INDICATOR 1
         MVC   F_IND1(2),0(R7)    MOVE PARM1 INDICATOR TO LOCAL STORAGE
         LH    R7,F_IND1          MOVE PARM1 INDICATOR INTO R7
         LTR   R7,R7              CHECK IF IT IS NEGATIVE
         BM    NULLIN             IF SO, PARM1 IS NULL
         L     R7,16(R1)          GET POINTER TO INDICATOR 2
         MVC   F_IND2(2),0(R7)    MOVE PARM2 INDICATOR TO LOCAL STORAGE
         LH    R7,F_IND2          MOVE PARM2 INDICATOR INTO R7
         LTR   R7,R7              CHECK IF IT IS NEGATIVE
         BM    NULLIN             IF SO, PARM2 IS NULL
    .
    .
    .
         L     R7,8(R1)           GET ADDRESS OF AREA FOR RESULT
NULLIN   MVC   0(9,R7),RESULT     MOVE A VALUE INTO RESULT AREA
         L     R7,20(R1)          GET ADDRESS OF AREA FOR RESULT IND
         MVC   0(2,R7),=H'0'      MOVE A VALUE INTO INDICATOR AREA
    .
    .
    .
         CEETERM  RC=0
*******************************************************************
*  VARIABLE DECLARATIONS AND EQUATES                             *
*******************************************************************
R1       EQU   1                  REGISTER 1
R7       EQU   7                  REGISTER 7
PPA      CEEPPA ,                 CONSTANTS DESCRIBING THE CODE BLOCK
         LTORG ,                  PLACE LITERAL POOL HERE
PROGAREA DSECT
         ORG   *+CEEDSASZ         LEAVE SPACE FOR DSA FIXED PART
PARM1    DS    F                  PARAMETER 1
PARM2    DS    F                  PARAMETER 2
RESULT   DS    CL9                RESULT
F_IND1   DS    H                  INDICATOR FOR PARAMETER 1
F_IND2   DS    H                  INDICATOR FOR PARAMETER 2
F_INDR   DS    H                  INDICATOR FOR RESULT

PROGSIZE EQU   *-PROGAREA
         CEEDSA ,                 MAPPING OF THE DYNAMIC SAVE AREA
         CEECAA ,                 MAPPING OF THE COMMON ANCHOR AREA
         END   MYMAIN
```

Figure 98. How an assembler language user-defined function receives parameters

C or C++:

For C or C++ user-defined functions, the conventions for passing parameters are different for main programs and subprograms.

For subprograms, you pass the parameters directly. For main programs, you use the standard argc and argv variables to access the input and output parameters:

- The argv variable contains an array of pointers to the parameters that are passed to the user-defined function. All string parameters that are passed back to DB2 must be null terminated.
  - argv[0] contains the address of the load module name for the user-defined function.
  - argv[1] through argv[*n*] contain the addresses of parameters 1 through *n*.
- The argc variable contains the number of parameters that are passed to the external user-defined function, including argv[0].

Figure 99 shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result.

```
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
  int argc;
  char *argv[];
 {
   /**************************************************/
   /* Assume that the user-defined function invocation*/
   /* included 2 input  parameters in the parameter   */
   /* list.  Also assume that the definition includes */
   /* the SCRATCHPAD, FINAL CALL, and DBINFO options, */
   /* so DB2 passes the scratchpad, calltype, and     */
   /* dbinfo parameters.                              */
   /* The argv vector contains these entries:         */
   /*      argv[0]        1    load module name       */
   /*      argv[1-2]      2    input parms            */
   /*      argv[3]        1    result parm            */
   /*      argv[4-5]      2    null indicators        */
   /*      argv[6]        1    result null indicator  */
   /*      argv[7]        1    SQLSTATE variable       */
   /*      argv[8]        1    qualified func name    */
   /*      argv[9]        1    specific func name     */
   /*      argv[10]       1    diagnostic string      */
   /*      argv[11]       1    scratchpad             */
   /*      argv[12]       1    call type              */
   /*      argv[13]     + 1    dbinfo                 */
   /*                  ------                         */
   /*                   14    for the argc variable   */
   /**************************************************/
   if argc<>14
   {
:
:
:
   /********************************************************/
   /* This section would contain the code executed if the   */
   /* user-defined function is invoked with the wrong number */
   /* of parameters.                                         */
   /********************************************************/
   }
```

Figure 99. How a C or C++ user-defined function that is written as a main program receives parameters (Part 1 of 2)

```
/****************************************************/
/* Assume the first parameter is an integer.        */
/* The code below shows how to copy the integer     */
/* parameter into the application storage.          */
/****************************************************/
int parm1;
parm1 = *(int *) argv[1];

/****************************************************/
/* Access the null indicator for the first         */
/* parameter on the invoked user-defined function  */
/* as follows:                                      */
/****************************************************/
short int ind1;
ind1 = *(short int *) argv[4];

/****************************************************/
/* Use the expression below to assign              */
/* 'xxxxx' to the SQLSTATE returned to caller on    */
/* the SQL statement that contains the invoked      */
/* user-defined function.                           */
/****************************************************/
strcpy(argv[7],"xxxxx/0");

/****************************************************/
/* Obtain the value of the qualified function      */
/* name with this expression.                       */
/****************************************************/
char f_func[28];
strcpy(f_func,argv[8]);
/****************************************************/
/* Obtain the value of the specific function       */
/* name with this expression.                       */
/****************************************************/
char f_spec[19];
strcpy(f_spec,argv[9]);

/****************************************************/
/* Use the expression below to assign              */
/* 'yyyyyyyy' to the diagnostic string returned    */
/* in the SQLCA associated with the invoked         */
/* user-defined function.                           */
/****************************************************/
strcpy(argv[10],"yyyyyyyy/0");

/****************************************************/
/* Use the expression below to assign the          */
/* result of the function.                          */
/****************************************************/
char l_result[11];
strcpy(argv[3],l_result);


   :
   :
}
```

Figure 99. How a C or C++ user-defined function that is written as a main program receives parameters (Part 2 of 2)

Figure 100 on page 267 shows the parameter conventions for a user-defined scalar function written as a C subprogram that receives 2 parameters and returns one result.

```
                    #pragma runopts(plist(os))
                    #include <stdlib.h>
                    #include <stdio.h>
                    #include <string.h>

                    #define SQLUDF_ASCII              0     /* ASCII         */
                    #define SQLUDF_EBCDIC             1     /* EBCDIC        */
                    #define SQLUDF_UNICODE            2     /* UNICODE       */
                    struct sqludf_scratchpadstruct sqludf_scratchpad
                    {
                      unsigned long  length;      /* length of scratchpad data */
                      char  data[SQLUDF_SCRATCHPAD_LEN];   /* scratchpad data */
                    };
                    struct sqludf_dbinfo
                    {
                      unsigned short dbnamelen;   /* database name length      */
                      unsigned char  dbname[128]; /* database name             */
                      unsigned short authidlen;   /* appl auth id  length      */
                      unsigned char  authid[128]; /* appl authorization ID     */
  |                   struct db2_cdpg
  |                   {
  |                     struct db2_ccsids
  |                     {
  |                       unsigned long db2_sbcs;
  |                       unsigned long db2_dbcs;
  |                       unsigned long db2_mixed;
  |                     } db2_ccsids_t[3];
  |
  |                     unsigned long db2_encoding_scheme;
  |                     unsigned char reserved[8];
  |                   };
                      unsigned short tbqualiflen;    /* table qualifier length */
                      unsigned char  tbqualif[128];  /* table qualifer name    */
                      unsigned short tbnamelen;      /* table name length      */
                      unsigned char  tbname[128];    /* table name             */
                      unsigned short colnamelen;     /* column name length     */
                      unsigned char  colname[128];   /* column name            */
                      unsigned char  relver[8]; /* Database release & version  */
                      unsigned long  platform;  /* Database platform           */
                      unsigned short numtfcol;  /* # of Tab Fun columns used   */
                      unsigned char  reserv1[24];  /* reserved                 */
                      unsigned short *tfcolnum;    /* table fn column list      */
                      unsigned short *appl_id; /* LUWID for DB2 connection      */
                      unsigned char  reserv2[20];  /* reserved                  */
                    };
```

*Figure 100. How a C language user-defined function that is written as a subprogram receives
parameters (Part 1 of 2)*

```
                void myfunc(long *parm1, char parm2[11], char result[11],
                            short *f_ind1, short *f_ind2, short *f_indr,
                            char udf_sqlstate[6], char udf_fname[138],
                            char udf_specname[129], char udf_msgtext[71],
                            struct sqludf_scratchpad *udf_scratchpad,
                            long *udf_call_type,
                            struct sql_dbinfo *udf_dbinfo);
                {
                    /****************************************************/
                    /* Declare local copies of parameters             */
                    /****************************************************/
                    int l_p1;
                    char l_p2[11];
                    short int l_ind1;
                    short int l_ind2;
                    char ludf_sqlstate[6];     /* SQLSTATE                */
                    char ludf_fname[138];       /* function name          */
                    char ludf_specname[129];   /* specific function name */
                    char ludf_msgtext[71]      /* diagnostic message text*/
                    sqludf_scratchpad *ludf_scratchpad; /* scratchpad    */
                    long *ludf_call_type;      /* call type              */
                    sqludf_dbinfo *ludf_dbinfo /* dbinfo                 */
                    /****************************************************/
                    /* Copy each of the parameters in the parameter   */
                    /* list into a local variable to demonstrate      */
                    /* how the parameters can be referenced.          */
                    /****************************************************/

                    l_p1 = *parm1;
                    strcpy(l_p2,parm2);
                    l_ind1 = *f_ind1;
                    l_ind1 = *f_ind2;
                    strcpy(ludf_sqlstate,udf_sqlstate);
                    strcpy(ludf_fname,udf_fname);
                    strcpy(ludf_specname,udf_specname);
                    l_udf_call_type = *udf_call_type;
                    strcpy(ludf_msgtext,udf_msgtext);
                    memcpy(&ludf_scratchpad,udf_scratchpad,sizeof(ludf_scratchpad));
                    memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));

                      .
                      .
                      .
                }
```

Figure 100. How a C language user-defined function that is written as a subprogram receives parameters (Part 2 of 2)

Figure 101 on page 269 shows the parameter conventions for a user-defined scalar function that is written as a C++ subprogram that receives two parameters and returns one result. This example demonstrates that you must use an extern "C" modifier to indicate that you want the C++ subprogram to receive parameters according to the C linkage convention. This modifier is necessary because the CEEPIPI CALL_SUB interface, which DB2 uses to call the user-defined function, passes parameters using the C linkage convention.

```
#pragma runopts(plist(os))
#include <stdlib.h>
#include <stdio.h>
#define SQLUDF_ASCII            0    /* ASCII          */
#define SQLUDF_EBCDIC           1    /* EBCDIC         */
#define SQLUDF_UNICODE          2    /* UNICODE        */
struct sqludf_scratchpad
{
  unsigned long  length;     /* length of scratchpad data */
  char  data[SQLUDF_SCRATCHPAD_LEN];  /* scratchpad data */
};
struct sqludf_dbinfo
{
  unsigned short dbnamelen;   /* database name length      */
  unsigned char  dbname[128]; /* database name             */
  unsigned short authidlen;   /* appl auth id  length      */
  unsigned char  authid[128]; /* appl authorization ID     */
    struct db2_cdpg
    {
      struct db2_ccsids
      {
         unsigned long db2_sbcs;
         unsigned long db2_dbcs;
         unsigned long db2_mixed;
      } db2_ccsids_t[3];

      unsigned long db2_encoding_scheme;
      unsigned char reserved[8];
    };
  unsigned short tbquaiflen;     /* table qualifier length */
  unsigned char  tbqualif[128];  /* table qualifer name    */
  unsigned short tbnamelen;      /* table name length      */
  unsigned char  tbname[128];    /* table name             */
  unsigned short colnamelen;     /* column name length     */
  unsigned char  colname[128];   /* column name            */
  unsigned char  relver[8]; /* Database release & version  */
  unsigned long  platform;  /* Database platform           */
  unsigned short numtfcol;  /* # of Tab Fun columns used   */
  unsigned char  reserv1[24];  /* reserved                 */
  unsigned short *tfcolnum;    /* table fn column list      */
  unsigned short *appl_id; /* LUWID for DB2 connection      */
  unsigned char  reserv2[20];  /* reserved                 */
};
extern "C" void myfunc(long *parm1, char parm2[11],
          char result[11], short *f_ind1, short *f_ind2, short *f_indr,
          char udf_sqlstate[6], char udf_fname[138],
          char udf_specname[129], char udf_msgtext[71],
          struct sqludf_scratchpad *udf_scratchpad,
          long *udf_call_type,
          struct sql_dbinfo *udf_dbinfo);
```

Figure 101. How a C++ user-defined function that is written as a subprogram receives parameters (Part 1 of 2)

```
{
  /**************************************************/
  /* Define local copies of parameters.           */
  /**************************************************/
  int l_p1;
  char l_p2[11];
  short int l_ind1;
  short int l_ind2;
  char ludf_sqlstate[6];      /* SQLSTATE             */
  char ludf_fname[138];       /* function name        */
  char ludf_specname[129];    /* specific function name */
  char ludf_msgtext[71]       /* diagnostic message text*/
  sqludf_scratchpad *ludf_scratchpad; /* scratchpad      */
  long *ludf_call_type;       /* call type            */
  sqludf_dbinfo *ludf_dbinfo /* dbinfo               */
  /**************************************************/
  /* Copy each of the parameters in the parameter   */
  /* list into a local variable to demonstrate      */
  /* how the parameters can be referenced.          */
  /**************************************************/
  l_p1 = *parm1;
  strcpy(l_p2,parm2);
  l_ind1 = *f_ind1;
  l_ind1 = *f_ind2;
  strcpy(ludf_sqlstate,udf_sqlstate);
  strcpy(ludf_fname,udf_fname);
  strcpy(ludf_specname,udf_specname);
  l_udf_call_type = *udf_call_type;
  strcpy(ludf_msgtext,udf_msgtext);
  memcpy(&ludf_scratchpad,udf_scratchpad,sizeof(ludf_scratchpad));
  memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));

  .
  .
  .
}
```

Figure 101. How a C++ user-defined function that is written as a subprogram receives parameters (Part 2 of 2)

COBOL: Figure 102 on page 271 shows the parameter conventions for a user-defined table function that is written as a main program that receives two parameters and returns two results. For a COBOL user-defined function that is a subprogram, the conventions are the same.

```
CBL APOST,RES,RENT
      IDENTIFICATION DIVISION.
   .
   .
   .
      DATA DIVISION.
   .
   .
   .
      LINKAGE SECTION.
     **********************************************************
     * Declare each of the parameters                        *
     **********************************************************
      01  UDFPARM1 PIC S9(9) USAGE COMP.
      01  UDFPARM2 PIC X(10).
   .
   .
   .
     **********************************************************
     * Declare these variables for result parameters         *
     **********************************************************
      01  UDFRESULT1 PIC X(10).
      01  UDFRESULT2 PIC X(10).
   .
   .
   .
     **********************************************************
     * Declare a null indicator for each parameter           *
     **********************************************************
      01  UDF-IND1 PIC S9(4) USAGE COMP.
      01  UDF-IND2 PIC S9(4) USAGE COMP.
   .
   .
   .
     **********************************************************
     * Declare a null indicator for result parameter         *
     **********************************************************
      01  UDF-RIND1 PIC S9(4) USAGE COMP.
      01  UDF-RIND2 PIC S9(4) USAGE COMP.
   .
   .
   .
     **********************************************************
     * Declare the SQLSTATE that can be set by the           *
     * user-defined function                                 *
     **********************************************************
      01  UDF-SQLSTATE PIC X(5).
     **********************************************************
     * Declare the qualified function name                   *
     **********************************************************
      01  UDF-FUNC.
          49 UDF-FUNC-LEN PIC 9(4) USAGE BINARY.
          49 UDF-FUNC-TEXT PIC X(137).
     **********************************************************
     * Declare the specific function name                    *
     **********************************************************
      01  UDF-SPEC.
          49 UDF-SPEC-LEN PIC 9(4) USAGE BINARY.
          49 UDF-SPEC-TEXT PIC X(128).
```

Figure 102. How a COBOL user-defined function receives parameters (Part 1 of 3)

```
          ***********************************************************
          * Declare SQL diagnostic message token                  *
          ***********************************************************
           01  UDF-DIAG.
               49 UDF-DIAG-LEN PIC 9(4) USAGE BINARY.
               49 UDF-DIAG-TEXT PIC X(70).
          ***********************************************************
          * Declare the scratchpad                                *
          ***********************************************************
           01  UDF-SCRATCHPAD.
               49 UDF-SPAD-LEN PIC 9(9) USAGE BINARY.
               49 UDF-SPAD-TEXT PIC X(100).
          ***********************************************************
          * Declare the call type                                 *
          ***********************************************************
           01  UDF-CALL-TYPE PIC 9(9) USAGE BINARY.
          *****************************************************
          * CONSTANTS FOR DB2-EBCODING-SCHEME.               *
          *****************************************************
           77 SQLUDF-ASCII PIC 9(9) VALUE 1.
           77 SQLUDF-EBCDIC PIC 9(9) VALUE 2.
           77 SQLUDF-UNICODE PIC 9(9) VALUE 3.
          ***********************************************************
          * Declare the DBINFO structure
          ***********************************************************
           01  UDF-DBINFO.
          *       Location length and name
               02 UDF-DBINFO-LOCATION.
                  49 UDF-DBINFO-LLEN PIC 9(4) USAGE BINARY.
                  49 UDF-DBINFO-LOC  PIC X(128).
          *       Authorization ID length and name
               02 UDF-DBINFO-AUTHORIZATION.
                  49 UDF-DBINFO-ALEN PIC 9(4) USAGE BINARY.
                  49 UDF-DBINFO-AUTH PIC X(128).
          *       CCSIDs for DB2 for OS/390
               02 UDF-DBINFO-CCSID PIC X(48).
               02 UDF-DBINFO-CDPG REDEFINES UDF-DBINFO-CCSID.
                  03 DB2-CCSIDS OCCURS 3 TIMES.
                     04 DB2-SBCS   PIC 9(9) USAGE BINARY.
                     04 DB2-DBCS   PIC 9(9) USAGE BINARY.
                     04 DB2-MIXED  PIC 9(9) USAGE BINARY.
                  03 DB2-ENCODING-SCHEME   PIC 9(9) USAGE BINARY.
                  03 DB2-CCSID-RESERVED    PIC X(8).
          *       Schema length and name
               02 UDF-DBINFO-SCHEMA0.
                  49 UDF-DBINFO-SLEN PIC 9(4) USAGE BINARY.
                  49 UDF-DBINFO-SCHEMA PIC X(128).
          *       Table length and name
               02 UDF-DBINFO-TABLE0.
                  49 UDF-DBINFO-TLEN PIC 9(4) USAGE BINARY.
                  49 UDF-DBINFO-TABLE  PIC X(128).
          *       Column length and name
               02 UDF-DBINFO-COLUMN0.
                  49 UDF-DBINFO-CLEN PIC 9(4) USAGE BINARY.
                  49 UDF-DBINFO-COLUMN  PIC X(128).
          *       DB2 release level
               02 UDF-DBINFO-VERREL PIC X(8).
```

Figure 102. How a COBOL user-defined function receives parameters (Part 2 of 3)

```
      *       Unused
       02 FILLER          PIC X(2).
      *       Database Platform
       02 UDF-DBINFO-PLATFORM PIC 9(9) USAGE BINARY.
      *       # of entries in Table Function column list
       02 UDF-DBINFO-NUMTFCOL PIC 9(4) USAGE BINARY.
      *       reserved
       02 UDF-DBINFO-RESERV1 PIC X(24).
      *       Unused
       02 FILLER          PIC X(2).
      *       Pointer to Table Function column list
       02 UDF-DBINFO-TFCOLUMN PIC 9(9) USAGE BINARY.
      *       Pointer to Application ID
       02 UDF-DBINFO-APPLID  PIC 9(9) USAGE BINARY.
      *       reserved
       02 UDF-DBINFO-RESERV2 PIC X(20).
      *
   PROCEDURE DIVISION USING UDFPARM1, UDFPARM2, UDFRESULT1,
                            UDFRESULT2, UDF-IND1, UDF-IND2,
                            UDF-RIND1, UDF-RIND2,
                            UDF-SQLSTATE, UDF-FUNC, UDF-SPEC,
                            UDF-DIAG, UDF-SCRATCHPAD,
                            UDF-CALL-TYPE, UDF-DBINFO.
```

Figure 102. How a COBOL user-defined function receives parameters (Part 3 of 3)

PL/I: Figure 103 on page 274 shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For a PL/I user-defined function that is a subprogram, the conventions are the same.

```
*PROCESS SYSTEM(MVS);
 MYMAIN: PROC(UDF_PARM1, UDF_PARM2, UDF_RESULT,
              UDF_IND1, UDF_IND2, UDF_INDR,
              UDF_SQLSTATE, UDF_NAME, UDF_SPEC_NAME,
              UDF_DIAG_MSG, UDF_SCRATCHPAD,
              UDF_CALL_TYPE, UDF_DBINFO)
          OPTIONS(MAIN NOEXECOPS REENTRANT);

 DCL UDF_PARM1 BIN FIXED(31);    /* first parameter          */
 DCL UDF_PARM2 CHAR(10);         /* second parameter         */
 DCL UDF_RESULT CHAR(10);        /* result parameter         */
 DCL UDF_IND1 BIN FIXED(15);     /* indicator for 1st parm   */
 DCL UDF_IND2 BIN FIXED(15);     /* indicator for 2nd parm   */
 DCL UDF_INDR BIN FIXED(15);     /* indicator for result     */
 DCL UDF_SQLSTATE CHAR(5);       /* SQLSTATE returned to DB2  */
 DCL UDF_NAME CHAR(137) VARYING; /* Qualified function name    */
 DCL UDF_SPEC_NAME CHAR(128) VARYING; /* Specific function name */
 DCL UDF_DIAG_MSG CHAR(70) VARYING; /* Diagnostic string       */
 DCL 01 UDF_SCRATCHPAD           /* Scratchpad               */
     03 UDF_SPAD_LEN BIN FIXED(31),
     03 UDF_SPAD_TEXT CHAR(100);
 DCL UDF_CALL_TYPE BIN FIXED(31); /* Call Type               */
 DCL DBINFO PTR;
 /* CONSTANTS FOR DB2_ENCODING_SCHEME */
 DCL SQLUDF_ASCII BIN FIXED(15) INIT(1);
 DCL SQLUDF_EBCDIC BIN FIXED(15) INIT(2);
 DCL SQLUDF_MIXED BIN FIXED(15) INIT(3);
 DCL 01 UDF_DBINFO BASED(DBINFO),         /* Dbinfo          */
        03 UDF_DBINFO_LLEN BIN FIXED(15),    /* location length */
        03 UDF_DBINFO_LOC  CHAR(128),        /* location name   */
        03 UDF_DBINFO_ALEN BIN FIXED(15),    /* auth ID  length */
        03 UDF_DBINFO_AUTH CHAR(128),        /* authorization ID */
        03 UDF_DBINFO_CDPG,          /* CCSIDs for DB2 for OS/390*/
          05 DB2_CCSIDS(3),
            07 R1               BIN FIXED(15), /* Reserved         */
            07 DB2_SBCS  BIN FIXED(15),      /* SBCS CCSID  */
            07 R2               BIN FIXED(15), /* Reserved        */
            07 DB2_DBCS  BIN FIXED(15),      /* DBCS CCSID  */
            07 R3               BIN FIXED(15), /* Reserved        */
            07 DB2_MIXED  BIN FIXED(15),     /* MIXED CCSID  */
          05 DB2_ENCODING_SCHEME BIN FIXED(31),
          05 DB2_CCSID_RESERVED CHAR(8),
```

Figure 103. How a PL/I user-defined function receives parameters (Part 1 of 2)

```
        03 UDF_DBINFO_SLEN BIN FIXED(15),    /* schema length   */
        03 UDF_DBINFO_SCHEMA CHAR(128),      /* schema name     */
        03 UDF_DBINFO_TLEN BIN FIXED(15),    /* table length    */
        03 UDF_DBINFO_TABLE  CHAR(128),      /* table name      */
        03 UDF_DBINFO_CLEN BIN FIXED(15),    /* column length   */
        03 UDF_DBINFO_COLUMN  CHAR(128),     /* column name     */
        03 UDF_DBINFO_RELVER  CHAR(8),       /* DB2 release level */
        03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform*/
        03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF cols used*/
        03 UDF_DBINFO_RESERV1 CHAR(24),      /* reserved        */
        03 UDF_DBINFO_TFCOLUMN PTR,      /* -> table fun col list */
        03 UDF_DBINFO_APPLID   PTR,      /* -> application id     */
        03 UDF_DBINFO_RESERV2 CHAR(20);      /* reserved        */

    ⋮
```

Figure 103. How a PL/I user-defined function receives parameters (Part 2 of 2)

Using special registers in a user-defined function

You can use all special registers in a user-defined function. However, you can modify only some of those special registers. After a user-defined function completes, DB2 restores all special registers to the values they had before invocation.

Table 35 shows information you need when you use special registers in a user-defined function.

Table 35. Characteristics of special registers in a user-defined function

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET statement to modify?
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the user-defined function package[1]	The value of bind option ENCODING for the user-defined function package[1]	Yes
CURRENT DATE	New value for each SQL statement in the user-defined function package[2]	New value for each SQL statement in the user-defined function package[2]	Not applicable[5]
CURRENT DEGREE	Inherited from invoker[3]	The value of field CURRENT DEGREE on installation panel DSNTIP4	Yes
CURRENT LOCALE LC_CTYPE	Inherited from invoker	The value of field CURRENT DEGREE on installation panel DSNTIP4	Yes
CURRENT MEMBER	New value for each SET *host-variable*=CURRENT MEMBER statement	New value for each SET *host-variable*=CURRENT MEMBER statement	No
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the user-defined function package or inherited from invoker[6]	The value of bind option OPTHINT for the user-defined function package	Yes
CURRENT PACKAGESET	Inherited from invoker[4]	Inherited from invoker[4]	Yes
CURRENT PATH	The value of bind option PATH for the user-defined function package or inherited from invoker[6]	The value of bind option PATH for the user-defined function package	Yes
CURRENT PRECISION	Inherited from invoker	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT RULES	Inherited from invoker	The value of bind option SQLRULES for the user-defined function package	Yes
CURRENT SERVER	Inherited from invoker	Inherited from invoker	Yes

Table 35. Characteristics of special registers in a user-defined function (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET statement to modify?
CURRENT SQLID	The primary authorization ID of the application process or inherited from invoker[7]	The primary authorization ID of the application process	Yes[8]
CURRENT TIME	New value for each SQL statement in the user-defined function package[2]	New value for each SQL statement in the user-defined function package[2]	Not applicable[5]
CURRENT TIMESTAMP	New value for each SQL statement in the user-defined function package[2]	New value for each SQL statement in the user-defined function package[2]	Not applicable[5]
CURRENT TIMEZONE	Inherited from invoker	Inherited from invoker	Not applicable[5]
CURRENT USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable[5]

Notes:

1. If the ENCODING bind option is not specified, the initial value is the value that was specified in field APPLICATION ENCODING of installation panel DSNTIPF.

2. If the user-defined function is invoked within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the function package.

3. DB2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, DB2 ignores the CURRENT DEGREE value.

4. If the user-defined function definer specifies a value for COLLID in the CREATE FUNCTION statement, DB2 sets CURRENT PACKAGESET to the value of COLLID.

5. Not applicable because no SET statement exists for the special register.

6. If a program within the scope of the invoking program issues a SET statement for the special register before the user-defined function is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the user-defined function package.

7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the user-defined function is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.

8. If the user-defined function package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed but does not affect the authorization ID that is used for the dynamic SQL statements in the user-defined function package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements. See "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415 for more information on DYNAMICRULES values and authorization IDs.

Using a scratchpad in a user-defined function

You can use a scratchpad to save information between invocations of a user-defined function. To indicate that a scratchpad should be allocated when the

user-defined function executes, the function definer specifies the SCRATCHPAD parameter in the CREATE FUNCTION statement.

The scratchpad consists of a 4-byte length field, followed by the scratchpad area. The definer can specify the length of the scratchpad area in the CREATE FUNCTION statement. The specified length does not include the length field. The default size is 100 bytes. DB2 initializes the scratchpad for each function to binary zeros at the beginning of execution for each subquery of an SQL statement and does not examine or change the content thereafter. On each invocation of the user-defined function, DB2 passes the scratchpad to the user-defined function. You can therefore use the scratchpad to preserve information between invocations of a reentrant user-defined function.

Figure 104 on page 278 demonstrates how to enter information in a scratchpad for a user-defined function defined like this:

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  FENCED
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'UDFCTR';
```

The scratchpad length is not specified, so the scratchpad has the default length of 100 bytes, plus 4 bytes for the length field. The user-defined function increments an integer value and stores it in the scratchpad on each execution.

```
#pragma linkage(ctr,fetchable)
#include <stdlib.h>
#include <stdio.h>
/* Structure scr defines the passed scratchpad for function ctr */
  struct scr {
    long len;
    long countr;
    char not_used[96];
  };
/***************************************************************/
/* Function ctr:  Increments a counter and reports the value  */
/*                from the scratchpad.                         */
/*                                                             */
/*    Input:  None                                             */
/*    Output: INTEGER out     the value from the scratchpad    */
/***************************************************************/
void ctr(
  long *out,                    /* Output answer (counter)   */
  short *outnull,               /* Output null indicator     */
  char *sqlstate,               /* SQLSTATE                  */
  char *funcname,               /* Function name             */
  char *specname,               /* Specific function name    */
  char *mesgtext,               /* Message text insert       */
  struct scr *scratchptr)       /* Scratchpad                */
{
  *out = ++scratchptr->countr;  /* Increment counter and     */
                                /* copy to output variable   */
  *outnull = 0;                 /* Set output null indicator */
   return;
}
/* end of user-defined function ctr */
```

Figure 104. Example of coding a scratchpad in a user-defined function

Accessing transition tables in a user-defined function or stored procedure

When you write a user-defined function, external stored procedure, or SQL procedure that is invoked from a trigger, you might need access to transition tables for the trigger. This section describes how to access transition variables in a user-defined function, but the same techniques apply to a stored procedure.

To access transition tables in a user-defined function, use table locators, which are pointers to the transition tables. You declare table locators as input parameters in the CREATE FUNCTION statement using the TABLE LIKE *table-name* AS LOCATOR clause. See Chapter 5 of *DB2 SQL Reference* for more information.

The five basic steps to accessing transition tables in a user-defined function are:

1. Declare input parameters to receive table locators. You must define each parameter that receives a table locator as an unsigned 4-byte integer.
2. Declare table locators. You can declare table locators in assembler, C, C⁺⁺, COBOL, PL/I, and in an SQL procedure compound statement.. The syntax for declaring table locators in C, C⁺⁺, COBOL, and PL/I is described in "Chapter 9. Embedding SQL statements in host languages" on page 107. The syntax for declaring table locators in an SQL procedure is described in Chapter 6 of *DB2 SQL Reference*.
3. Declare a cursor to access the rows in each transition table.
4. Assign the input parameter values to the table locators.

5. Access rows from the transition tables using the cursors that are declared for the transition tables.

The following examples show how a user-defined function that is written in C, C++, COBOL, or PL/I accesses a transition table for a trigger. The transition table, NEWEMP, contains modified rows of the employee sample table. The trigger is defined like this:

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES (CHECKEMP(TABLE NEWEMPS));
  END;
```

The user-defined function definition looks like this:

```
CREATE FUNCTION CHECKEMP(TABLE LIKE EMP AS LOCATOR)
  RETURNS INTEGER
  EXTERNAL NAME 'CHECKEMP'
  PARAMETER STYLE DB2SQL
  LANGUAGE language;
```

Assembler: Figure 105 on page 280 shows how an assembler program accesses rows of transition table NEWEMPS.

```
CHECKEMP CSECT
         SAVE  (14,12)             ANY SAVE SEQUENCE
         LR    R12,R15             CODE ADDRESSABILITY
         USING CHECKEMP,R12        TELL THE ASSEMBLER
         LR    R7,R1               SAVE THE PARM POINTER
         USING PARMAREA,R7         SET ADDRESSABILITY FOR PARMS
         USING SQLDSECT,R8         ESTABLISH ADDRESSIBILITY TO SQLDSECT
         L     R6,PROGSIZE         GET SPACE FOR USER PROGRAM
         GETMAIN R,LV=(6)          GET STORAGE FOR PROGRAM VARIABLES
         LR    R10,R1              POINT TO THE ACQUIRED STORAGE
         LR    R2,R10              POINT TO THE FIELD
         LR    R3,R6               GET ITS LENGTH
         SR    R4,R4               CLEAR THE INPUT ADDRESS
         SR    R5,R5               CLEAR THE INPUT LENGTH
         MVCL  R2,R4               CLEAR OUT THE FIELD
         ST    R13,FOUR(R10)       CHAIN THE SAVEAREA PTRS
         ST    R10,EIGHT(R13)      CHAIN SAVEAREA FORWARD
         LR    R13,R10             POINT TO THE SAVEAREA
         USING PROGAREA,R13        SET ADDRESSABILITY
         ST    R6,GETLENTH         SAVE THE LENGTH OF THE GETMAIN
    .
    .
    .
***************************************************************
* Declare table locator host variable TRIGTBL                *
***************************************************************
TRIGTBL SQL TYPE IS TABLE LIKE EMP AS LOCATOR
***************************************************************
* Declare a cursor to retrieve rows from the transition      *
* table                                                      *
***************************************************************
         EXEC SQL DECLARE C1 CURSOR FOR                          X
                SELECT LASTNAME FROM TABLE(:TRIGTBL LIKE EMP)     X
                WHERE SALARY > 100000
***************************************************************
* Copy table locator for trigger transition table            *
***************************************************************
         L     R2,TABLOC           GET ADDRESS OF LOCATOR
         L     R2,0(0,R2)          GET LOCATOR VALUE
         ST    R2,TRIGTBL
         EXEC SQL OPEN C1
         EXEC SQL FETCH C1 INTO :NAME
    .
    .
    .
         EXEC SQL CLOSE C1
    .
    .
    .
```

Figure 105. How an assembler user-defined function accesses a transition table (Part 1 of 2)

```
PROGAREA DSECT                          WORKING STORAGE FOR THE PROGRAM
SAVEAREA DS    18F                      THIS ROUTINE'S SAVE AREA
GETLENTH DS    A                        GETMAIN LENGTH FOR THIS AREA
   ⋮
NAME     DS    CL24
   ⋮
         DS    0D
PROGSIZE EQU   *-PROGAREA               DYNAMIC WORKAREA SIZE
PARMAREA DSECT
TABLOC   DS    A                        INPUT PARAMETER FOR TABLE LOCATOR
   ⋮
         END   CHECKEMP
```

Figure 105. How an assembler user-defined function accesses a transition table (Part 2 of 2)

C or C⁺⁺: Figure 106 shows how a C or C⁺⁺ program accesses rows of transition table NEWEMPS.

```c
int CHECK_EMP(int trig_tbl_id)
{

  ⋮
 /**********************************************************/
 /* Declare table locator host variable trig_tbl_id  */
 /**********************************************************/
 EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE IS TABLE LIKE EMP AS LOCATOR trig_tbl_id;
   char name[25];
 EXEC SQL END DECLARE SECTION;

  ⋮
 /**********************************************************/
 /* Declare a cursor to retrieve rows from the transition  */
 /* table                                                  */
 /**********************************************************/
 EXEC SQL DECLARE C1 CURSOR FOR
   SELECT NAME FROM TABLE(:trig_tbl_id LIKE EMPLOYEE)
   WHERE SALARY > 100000;
 /**********************************************************/
 /* Fetch a row from transition table                     */
 /**********************************************************/
 EXEC SQL OPEN C1;
 EXEC SQL FETCH C1 INTO :name;

  ⋮
 EXEC SQL CLOSE C1;

  ⋮
}
```

Figure 106. How a C or C⁺⁺ user-defined function accesses a transition table

COBOL: Figure 107 on page 282 shows how a COBOL program accesses rows of transition table NEWEMPS.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID. CHECKEMP.
      ENVIRONMENT DIVISION.
      INPUT-OUTPUT SECTION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01  NAME   PIC X(24).
   :
      LINKAGE SECTION.
      **********************************************************
      * Declare table locator host variable TRIG-TBL-ID        *
      **********************************************************
      01  TRIG-TBL-ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
   :
      PROCEDURE DIVISION USING TRIG-TBL-ID.
   :
      **********************************************************
      * Declare cursor to retrieve rows from transition table *
      **********************************************************
      EXEC SQL DECLARE C1 CURSOR FOR
        SELECT NAME FROM TABLE(:TRIG-TBL-ID LIKE EMP)
        WHERE SALARY > 100000 END-EXEC.
      **********************************************************
      * Fetch a row from transition table                      *
      **********************************************************
      EXEC SQL OPEN C1 END-EXEC.
      EXEC SQL FETCH C1 INTO :NAME END-EXEC.
   :
      EXEC SQL CLOSE C1 END-EXEC.
   :
      PROG-END.
      GOBACK.
```

Figure 107. How a COBOL user-defined function accesses a transition table

PL/I: Figure 108 on page 283 shows how a PL/I program accesses rows of transition table NEWEMPS.

```
CHECK_EMP: PROC(TRIG_TBL_ID) RETURNS(BIN FIXED(31))
           OPTIONS(MAIN NOEXECOPS REENTRANT);
/****************************************************/
/* Declare table locator host variable TRIG_TBL_ID */
/****************************************************/
DECLARE TRIG_TBL_ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR;
DECLARE NAME CHAR(24);

  .
  .
  .
/****************************************************/
/* Declare a cursor to retrieve rows from the       */
/* transition table                                 */
/****************************************************/
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT NAME FROM TABLE(:TRIG_TBL_ID LIKE EMP)
  WHERE SALARY > 100000;
/****************************************************/
/* Retrieve rows from the transition table          */
/****************************************************/
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 INTO :NAME;

  .
  .
  .
EXEC SQL CLOSE C1;

  .
  .
  .
END CHECK_EMP;
```

Figure 108. How a PL/I user-defined function accesses a transition table

Preparing a user-defined function for execution

To prepare a user-defined function for execution, perform these steps:

1. Precompile the user-defined function program and bind the DBRM into a package.

   You need to do this only if your user-defined function contains SQL statements. You do not need to bind a plan for the user-defined function.

2. Compile the user-defined function program and link-edit it with Language Environment and RRSAF.

   You must compile the program with a compiler that supports Language Environment and link-edit the appropriate Language Environment components with the user-defined function. You must also link-edit the user-defined function with RRSAF.

   For the minimum compiler and Language Environment requirements for user-defined functions, see *DB2 Release Planning Guide*.

   The program preparation JCL samples DSNHASM, DSNHC, DSNHCPP, DSNHICOB, and DSNHPLI show you how to precompile, compile, and link-edit assembler, C, C⁺⁺, COBOL, and PL/I DB2 programs. If your DB2 subsystem has been installed to work with Language Environment, you can use this sample JCL when you prepare your user-defined functions. For object-oriented programs in C⁺⁺ or COBOL, see JCL samples DSNHCPP2 and DSNHICB2 for program preparation hints.

3. For a user-defined function that contains SQL statements, grant EXECUTE authority on the user-defined function package to the function definer.

Making a user-defined function reentrant

Compiling and link-editing your user-defined function as reentrant is recommended. (For an assembler program, you must also code the user-defined function to be reentrant.) Reentrant user-defined functions have the following advantages:

- The operating system does not need to load the user-defined function into storage every time the user-defined function is called.
- Multiple tasks in a WLM-established stored procedures address space can share a single copy of the user-defined function. This decreases the amount of virtual storage that is needed for code in the address space.

Preparing user-defined functions that contain multiple programs: If your user-defined function consists of several programs, you must bind each program that contains SQL statements into a separate package. The definer of the user-defined function must have EXECUTE authority for all packages that are part of the user-defined function.

When the primary program of a user-defined function calls another program, DB2 uses the CURRENT PACKAGESET special register to determine the collection to search for the called program's package. The primary program can change this collection ID by executing the statement SET CURRENT PACKAGESET. If the value of CURRENT PACKAGESET is blank, DB2 uses the method described in "The order of search" on page 413 to search for the package.

Determining the authorization ID for user-defined function invocation

If your user-defined function is invoked statically, the authorization ID under which the user-defined function is invoked is the owner of the package that contains the user-defined function invocation.

If the user-defined function is invoked dynamically, the authorization ID under which the user-defined function is invoked depends on the value of bind parameter DYNAMICRULES for the package that contains the function invocation.

While a user-defined function is executing, the authorization ID under which static SQL statements in the user-defined function package execute is the owner of the user-defined function package. The authorization ID under which dynamic SQL statements in the user-defined function package execute depends on the value of DYNAMICRULES with which the user-defined function package was bound.

DYNAMICRULES influences a number of features of an application program. For information on how DYNAMICRULES works, see "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415. For more information on the authorization needed to invoke and execute SQL statements in a user-defined function, see Chapter 5 of *DB2 SQL Reference* and Part 3 (Volume 1) of *DB2 Administration Guide*.

Preparing user-defined functions to run concurrently

Multiple user-defined functions and stored procedures can run concurrently, each under its own OS/390 task (TCB).

To maximize the number of user-defined functions and stored procedures that can run concurrently, follow these preparation recommendations:

- Ask the system administrator to set the region size parameter in the startup procedures for the WLM-established stored procedures address spaces to REGION=0. This lets an address space obtain the largest possible amount of storage below the 16-MB line.

- Limit storage required by application programs below the 16-MB line by:
  - Link-editing programs with the AMODE(31) and RMODE(ANY) attributes
  - Compiling COBOL programs with the RES and DATA(31) options
- Limit storage that is required by Language Environment by using these run-time options:

HEAP(,,ANY)	Allocates program heap storage above the 16-MB line
STACK(,,ANY,)	Allocates program stack storage above the 16-MB line
STORAGE(,,,4K)	Reduces reserve storage area below the line to 4 KB
BELOWHEAP(4K,,)	Reduces the heap storage below the line to 4 KB
LIBSTACK(4K,,)	Reduces the library stack below the line to 4 KB
ALL31(ON)	Causes all programs contained in the external user-defined function to execute with AMODE(31) and RMODE(ANY)

The definer can list these options as values of the RUN OPTIONS parameter of CREATE FUNCTION, or the system administrator can establish these options as defaults during Language Environment installation.

For example, the RUN OPTIONS option parameter could contain:
```
H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)
```
- Ask the system administrator to set the NUMTCB parameter for WLM-established stored procedures address spaces to a value greater than 1. This lets more than one TCB run in an address space. Be aware that setting NUMTCB to a value greater than 1 also reduces your level of application program isolation. For example, a bad pointer in one application can overwrite memory that is allocated by another application.

Testing a user-defined function

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where user-defined functions run. This section describes some alternative testing strategies.

CODE/370: You can use the CoOperative Development Environment/370 licensed program, which works with Language Environment, to test DB2 for OS/390 and z/OS user-defined functions written in any of the supported languages. You can use CODE/370 either interactively or in batch mode.

Using CODE/370 interactively: To test a user-defined function interactively using CODE/370, you must use the CODE/370 PWS Debug Tool on a workstation. You must also have CODE/370 installed on the OS/390 system where the user-defined function runs. To debug your user-defined function using the PWS Debug Tool:

1. Compile the user-defined function with the TEST option. This places information in the program that the Debug Tool uses.
2. Invoke the debug tool. One way to do that is to specify the Language Environment run-time TEST option. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run-time options is with the RUN OPTIONS parameter of CREATE FUNCTION or ALTER

FUNCTION. See "Components of a user-defined function definition" on page 244 for more information on the RUN OPTIONS parameter.

For example, suppose that you code this option:

```
TEST(ALL,*,PROMPT,JBJONES%SESSNA:)
```

The parameter values cause the following things to happen:

ALL
> The Debug Tool gains control when an attention interrupt, abend, or program or Language Environment condition of Severity 1 and above occurs.

* Debug commands will be entered from the terminal.

PROMPT
> The Debug Tool is invoked immediately after Language Environment initialization.

JBJONES%SESSNA:
> CODE/370 initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.

3. If you want to save the output from your debugging session, issue a command that names a log file. For example, the following command starts logging to a file on the workstation called dbgtool.log.

```
SET LOG ON FILE dbgtool.log;
```

This should be the first command that you enter from the terminal or include in your commands file.

Using CODE/370 in batch mode: To test your user-defined function in batch mode, you must have the CODE/370 Mainframe Interface (MFI) Debug Tool installed on the OS/390 system where the user-defined function runs. To debug your user-defined function in batch mode using the MFI Debug Tool:

1. If you plan to use the Language Environment run-time TEST option to invoke CODE/370, compile the user-defined function with the TEST option. This places information in the program that the Debug Tool uses during a debugging session.

2. Allocate a log data set to receive the output from CODE/370. Put a DD statement for the log data set in the start-up procedure for the stored procedures address space.

3. Enter commands in a data set that you want CODE/370 to execute. Put a DD statement for that data set in the start-up procedure for the stored procedures address space. To define the data set that contains MFI Debug Tool commands to CODE/370, specify its data set name or DD name in the TEST run-time option. For example, this option tells CODE/370 to look for the commands in the data set that is associated with DD name TESTDD:

```
TEST(ALL,TESTDD,PROMPT,*)
```

The first command in the commands data set should be:

```
SET LOG ON FILE ddname;
```

This command directs output from your debugging session to the log data set you defined in step 2. For example, if you defined a log data set with DD name INSPLOG in the start-up procedure for the stored procedures address space, the first command should be:

```
SET LOG ON FILE INSPLOG;
```

4. Invoke the Debug Tool. Two possible methods are:
   - Specify the Language Environment run-time TEST option. The most convenient place to do that is in the RUN OPTIONS parameter of CREATE FUNCTION or ALTER FUNCTION.
   - Put CEETEST calls in the user-defined function source code. If you use this approach for an existing user-defined function, you must compile, link-edit, and bind the user-defined function again. Then you must issue the STOP FUNCTION SPECIFIC and START FUNCTION SPECIFIC commands to reload the user-defined function.

   You can combine the Language Environment run-time TEST option with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

   For more information on CODE/370, see *CoOperative Development Environment/370: Debug Tool*.

   Route debugging messages to SYSPRINT: You can include simple print statements in your user-defined function code that you route to SYSPRINT. Then use System Display and Search Facility (SDSF) to examine the SYSPRINT contents while the WLM-established stored procedure address space is running. You can serialize I/O by running the WLM-established stored procedure address space with NUMTCB=1.

   Driver applications: You can write a small driver application that calls the user-defined function as a subprogram and passes the parameter list for the user-defined function. You can then test and debug the user-defined function as a normal DB2 application under TSO. You can then use TSO TEST and other commonly used debugging tools.

   SQL INSERT: You can use SQL to insert debugging information into a DB2 table. This allows other machines in the network (such as a workstation) to easily access the data in the table using DRDA access.

   DB2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Implementing an SQL scalar function

An SQL scalar function is a user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL scalar function can return only one parameter. You specify the SQL expression in the RETURN clause of the CREATE FUNCTION statement. The value of the SQL expression must be compatible with the data type of the parameter in the RETURNS clause.

See "Defining a user-defined function" on page 244 and Chapter 5 of *DB2 SQL Reference* for a description of the parameters that you can specify in the CREATE FUNCTION statement for an SQL scalar function.

To prepare an SQL scalar function for execution, you execute the CREATE FUNCTION statement, either statically or dynamically.

Example: Creating an SQL scalar function: Create an SQL scalar function that
returns the tangent of the input value. Use the built-in SIN and COS functions to
calculate the tangent.

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X);
```

Invoking a user-defined function

You can invoke a sourced or external user-defined scalar function in an SQL
statement wherever you use an expression. For a table function, you can invoke the
user-defined function only in the FROM clause of a SELECT statement. The
invoking SQL statement can be in a stand-alone program, a stored procedure, a
trigger body, or another user-defined function.

See the following sections for details you should know before you invoke a
user-defined function:
*   "Syntax for user-defined function invocation"
*   "Ensuring that DB2 executes the intended user-defined function" on page 289
*   "Casting of user-defined function arguments" on page 295
*   "What happens when a user-defined function abnormally terminates" on page 296

Syntax for user-defined function invocation

Use the syntax shown in Figure 109 when you invoke a user-defined scalar
function:

```
►►──function-name──(─┬──────────┬─┬─────────────────────────────┬─)──►◄
                     ├─ALL──────┤ │ ┌──────,────────────────────┐ │
                     └─DISTINCT─┘ └─▼─┬─expression───────────────┬┴┘
                                      └─TABLE──transition-table-name─┘
```
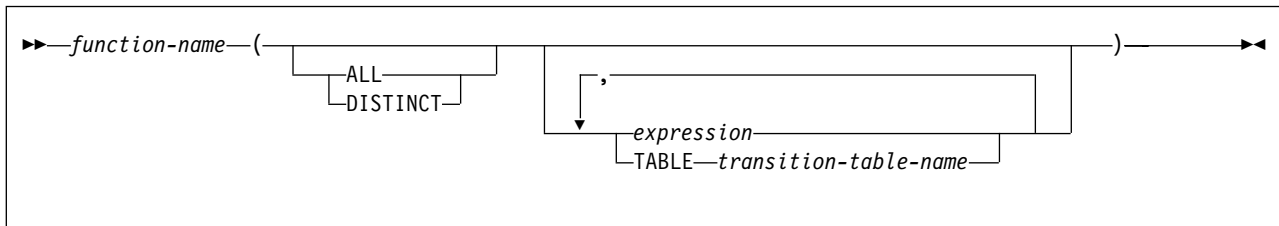
Figure 109. Syntax for user-defined scalar function invocation

Use the syntax shown in Figure 110 on page 289 when you invoke a table function:

```
►►─TABLE────(─function-name─(──────────────────────────────────────────────)─)─correlation-clause─►◄
                            │        ┌─,────────────────────┐                │
                            │        │                      │                │
                            └────────┼─expression───────────┼────────────────┘
                                     └─TABLE─transition-table-name─┘


correlation-clause:

          ┌─AS─┐
►►────────┴────┴──correlation-name────────────────────────────────────────►◄
                                     │        ┌─,──────────┐       │
                                     └─(──────┴─column-name─┴────)─┘
```
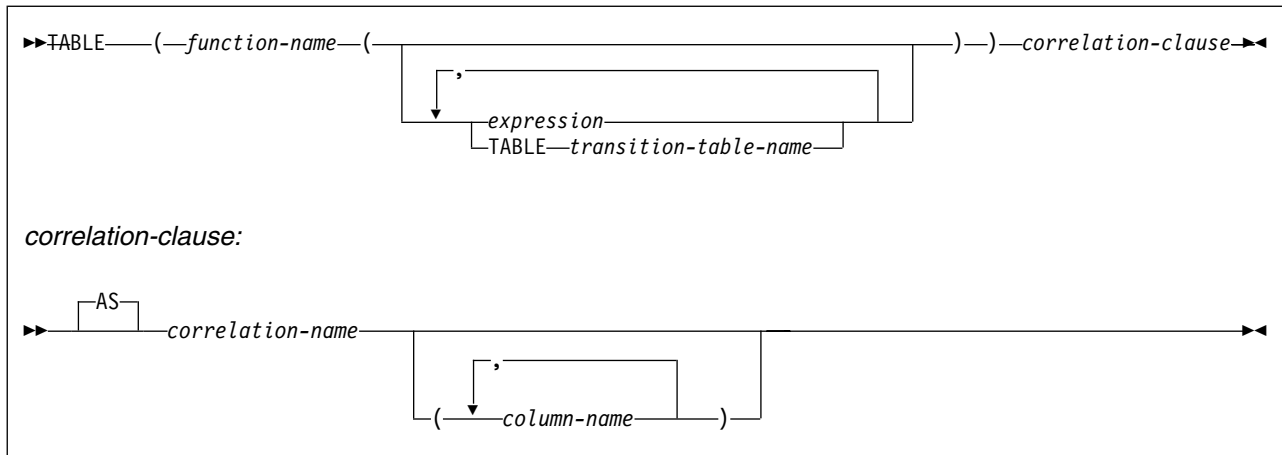
Figure 110. Syntax for table function invocation

See Chapter 2 of *DB2 SQL Reference* for more information about the syntax of user-defined function invocation.

Ensuring that DB2 executes the intended user-defined function

Several user-defined functions with the same name but different numbers or types of parameters can exist in a DB2 subsystem. Several user-defined functions with the same name can have the same number of parameters, as long as the data types of any of the first 30 parameters are different. In addition, several user-defined functions might have the same name as a built-in function. When you invoke a function, DB2 must determine which user-defined function or built-in function to execute. This process is known as *function resolution*. You need to understand DB2's function resolution process to ensure that you invoke the user-defined function that you want to invoke.

DB2 performs these steps for function resolution:

1. Determines if any function instances are candidates for execution. If no candidates exist, DB2 issues an SQL error message.

2. Compares the data types of the input parameters to determine which candidates fit the invocation best.

   DB2 does not compare data types for input parameters that are untyped parameter markers.

   For a qualified function invocation, if there are no parameter markers in the invocation, the result of the data type comparison is one best fit. That best fit is the choice for execution. If there are parameter markers in the invocation, there might be more than one best fit. DB2 issues an error if there is more than one best fit.

   For an unqualified function invocation, DB2 might find multiple best fits because the same function name with the same input parameters can exist in different schemas, or because there are parameter markers in the invocation.

3. If two or more candidates fit the unqualified function invocation equally well because the same function name with the same input parameters exists in different schemas, DB2 chooses the user-defined function whose schema name is earliest in the SQL path.

   For example, suppose functions SCHEMA1.X and SCHEMA2.X fit a function invocation equally well. Assume that the SQL path is:

   ```
   "SCHEMA2", "SYSPROC", "SYSIBM", "SCHEMA1", "SYSFUN"
   ```

Then DB2 chooses function SCHEMA2.X.

If two or more candidates fit the unqualified function invocation equally well because the function invocation contains parameter markers, DB2 issues an error.

The remainder of this section discusses details of the function resolution process and gives suggestions on how you can ensure that DB2 picks the right function.

How DB2 chooses candidate functions

An instance of a user-defined function is a candidate for execution only if it meets all of the following criteria:

- If the function name is qualified in the invocation, the schema of the function instance matches the schema in the function invocation.

  If the function name is unqualified in the invocation, the schema of the function instance matches a schema in the invoker's SQL path.

- The name of the function instance matches the name in the function invocation.

- The number of input parameters in the function instance matches the number of input parameters in the function invocation.

- The function invoker is authorized to execute the function instance.

- The type of each of the input parameters in the function invocation matches or is *promotable* to the type of the corresponding parameter in the function instance.

  If an input parameter in the function invocation is an untyped parameter marker, DB2 considers that parameter to be a match or promotable.

  For a function invocation that passes a transition table, the data type, length, precision, and scale of each column in the transition table must match exactly the data type, length, precision, and scale of each column of the table that is named in the function instance definition. For information on transition tables, see "Chapter 11. Using triggers for active data" on page 209.

- The create timestamp for a user-defined function must be older than the BIND or REBIND timestamp for the package or plan in which the user-defined function is invoked.

  If DB2 authorization checking is in effect, and DB2 performs an automatic rebind on a plan or package that contains a user-defined function invocation, any user-defined functions that were created after the original BIND or REBIND of the invoking plan or package are not candidates for execution.

  If you use an access control authorization exit routine, some user-defined functions that were not candidates for execution before the original BIND or REBIND of the invoking plan or package might become candidates for execution during the automatic rebind of the invoking plan or package. See Appendix B (Volume 2) of *DB2 Administration Guide* for information about function resolution with access control authorization exit routines.

  If a user-defined function is invoked during an automatic rebind, and that user-defined function is invoked from a trigger body and receives a transition table, then the form of the invoked function that DB2 uses for function selection includes only the columns of the transition table that existed at the time of the original BIND or REBIND of the package or plan for the invoking program.

  During an automatic rebind, DB2 does not consider built-in functions for function resolution if those built-in functions were introduced in a later release of DB2 than the release in which the BIND or REBIND of the invoking plan or package occurred.

  When you explicitly bind or rebind a plan or package, the plan or package receives a release dependency marker. When DB2 performs an automatic rebind

of a query that contains a function invocation, a built-in function is a candidate for function resolution only if the release dependency marker of the built-in function is the same as or lower than the release dependency marker of the plan or package that contains the function invocation.

To determine whether a data type is promotable to another data type, see Table 36. The first column lists data types in function invocations. The second column lists data types to which the types in the first column can be promoted, in order from best fit to worst fit. For example, suppose that in this statement, the data type of A is SMALLINT:

```
SELECT USER1.ADDTWO(A) FROM TABLEA;
```

Two instances of USER1.ADDTWO are defined: one with an input parameter of type INTEGER and one with an input parameter of type DECIMAL. Both function instances are candidates for execution because the SMALLINT type is promotable to either INTEGER or DECIMAL. However, the instance with the INTEGER type is a better fit because INTEGER is higher in the list than DECIMAL.

Table 36. Promotion of data types

Data type in function invocation	Possible fits (in best-to-worst order)
CHAR or GRAPHIC	CHAR or GRAPHIC VARCHAR or VARGRAPHIC CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC CLOB or DBCLOB
CLOB or DBCLOB[1]	CLOB or DBCLOB
BLOB[1]	BLOB
SMALLINT	SMALLINT INTEGER DECIMAL REAL DOUBLE
INTEGER	INTEGER DECIMAL REAL DOUBLE
DECIMAL	DECIMAL REAL DOUBLE
REAL[2]	REAL DOUBLE
DOUBLE[3]	DOUBLE
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
ROWID	ROWID
Distinct type	Distinct type with same name

Table 36. Promotion of data types (continued)

Data type in function invocation	Possible fits (in best-to-worst order)

Notes:

1. This promotion also applies if the parameter type in the invocation is a LOB locator for a LOB with this data type.
2. The FLOAT type with a length of less than 22 is equivalent to REAL.
3. The FLOAT type with a length of greater than or equal to 22 is equivalent to DOUBLE.

How DB2 chooses the best fit among candidate functions

More than one function instance might be a candidate for execution. In that case, DB2 determines which function instances are the best fit for the invocation by comparing parameter data types.

If the data types of all parameters in a function instance are the same as those in the function invocation, that function instance is a best fit. If no exact match exists, DB2 compares data types in the parameter lists from left to right, using this method:

1. DB2 compares the data types of the first parameter in the function invocation to the data type of the first parameter in each function instance.

   If the first parameter in the invocation is an untyped parameter marker, DB2 does not do the comparison.

2. For the first parameter, if one function instance has a data type that fits the function invocation better than the data types in the other instances, that function is a best fit. Table 36 on page 291 shows the possible fits for each data type, in best-to-worst order.

3. If the data types of the first parameter are the same for all function instances, or if the first parameter in the function invocation is an untyped parameter marker, DB2 repeats this process for the next parameter. DB2 continues this process for each parameter until it finds a best fit.

Example of function resolution: Suppose that a program contains the following statement:

```
SELECT FUNC(VCHARCOL,SMINTCOL,DECCOL) FROM T1;
```

In user-defined function FUNC, VCHARCOL has data type VARCHAR, SMINTCOL has data type SMALLINT, and DECCOL has data type DECIMAL. Also suppose that two function instances with the following definitions meet the criteria in "How DB2 chooses candidate functions" on page 290 and are therefore candidates for execution.

```
Candidate 1:
CREATE FUNCTION FUNC(VARCHAR(20),INTEGER,DOUBLE)
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'FUNC1'
  PARAMETER STYLE DB2SQL
  LANGUAGE COBOL;

Candidate 2:
CREATE FUNCTION FUNC(VARCHAR(20),REAL,DOUBLE)
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'FUNC2'
  PARAMETER STYLE DB2SQL
  LANGUAGE COBOL;
```

DB2 compares the data type of the first parameter in the user-defined function invocation to the data types of the first parameters in the candidate functions. Because the first parameter in the invocation has data type VARCHAR, and both

candidate functions also have data type VARCHAR, DB2 cannot determine the better candidate based on the first parameter. Therefore, DB2 compares the data types of the second parameters.

The data type of the second parameter in the invocation is SMALLINT. INTEGER, which is the data type of candidate 1, is a better fit to SMALLINT than REAL, which is the data type of candidate 2. Therefore, candidate 1 is DB2's choice for execution.

How you can simplify function resolution
When you use the following techniques, you can simplify function resolution:

- When you invoke a function, use the qualified name. This causes DB2 to search for functions only in the schema you specify. This has two advantages:
  - DB2 is less likely to choose a function that you did not intend to use. Several functions might fit the invocation equally well. DB2 picks the function whose schema name is earliest in the SQL path, which might not be the function you want.
  - The number of candidate functions is smaller, so DB2 takes less time for function resolution.
- Cast parameters in a user-defined function invocation to the types in the user-defined function definition. For example, if an input parameter for user-defined function FUNC is defined as DECIMAL(13,2), and the value you want to pass to the user-defined function is an integer value, cast the integer value to DECIMAL(13,2):

```
SELECT FUNC(CAST (INTCOL AS DECIMAL(13,2))) FROM T1;
```

- Avoid defining user-defined function numeric parameters as SMALLINT or REAL. Use INTEGER or DOUBLE instead. An invocation of a user-defined function defined with parameters of type SMALLINT or REAL must use parameters of the same types. For example, if user-defined function FUNC is defined with a parameter of type SMALLINT, only an invocation with a parameter of type SMALLINT resolves correctly. An invocation like this does not resolve to FUNC because the constant 123 is of type INTEGER, not SMALLINT:

```
SELECT FUNC(123) FROM T1;
```

- Avoid defining user-defined function string parameters with fixed-length string types. If you define a parameter with a fixed-length string type (CHAR or GRAPHIC), you can invoke the user-defined function only with a fixed-length string parameter. However, if you define the parameter with a varying-length string type (VARCHAR or VARGRAPHIC), you can invoke the user-defined function with either a fixed-length string parameter or a varying-length string parameter.

  If you must define parameters for a user-defined function as CHAR, and you call the user-defined function from a C program or SQL procedure, you need to cast the corresponding parameter values in the user-defined function invocation to CHAR to ensure that DB2 invokes the correct function. For example, suppose that a C program calls user-defined function CVRTNUM, which takes one input parameter of type CHAR(6). Also suppose that you declare host variable empnumbr as `char empnumbr[6]`. When you invoke CVRTNUM, cast empnumbr to CHAR:

```
UPDATE EMP
  SET EMPNO=CVRTNUM(CHAR(:empnumbr))
  WHERE EMPNO = :empnumbr;
```

Using DSN_FUNCTION_TABLE to see how DB2 resolves a function

You can use DB2's EXPLAIN tool to obtain information about how DB2 resolves functions. DB2 stores the information in a table called DSN_FUNCTION_TABLE, which you create. DB2 puts a row in DSN_FUNCTION_TABLE for each function that is referenced in an SQL statement when one of the following events occurs:

- You execute the SQL EXPLAIN statement on an SQL statement that contains user-defined function invocations.

- You run a program whose plan is bound with EXPLAIN(YES), and the program executes an SQL statement that contains user-defined function invocations.

Before you use EXPLAIN to obtain information about function resolution, create DSN_FUNCTION_TABLE. The table definition looks like this:

```
CREATE TABLE DSN_FUNCTION_TABLE
  (QUERYNO          INTEGER     NOT NULL WITH DEFAULT,
   QBLOCKNO         INTEGER     NOT NULL WITH DEFAULT,
   APPLNAME         CHAR(8)     NOT NULL WITH DEFAULT,
   PROGNAME         CHAR(8)     NOT NULL WITH DEFAULT,
   COLLID           CHAR(18)    NOT NULL WITH DEFAULT,
   GROUP_MEMBER     CHAR(8)     NOT NULL WITH DEFAULT,
   EXPLAIN_TIME     TIMESTAMP   NOT NULL WITH DEFAULT,
   SCHEMA_NAME      CHAR(8)     NOT NULL WITH DEFAULT,
   FUNCTION_NAME    CHAR(18)    NOT NULL WITH DEFAULT,
   SPEC_FUNC_NAME   CHAR(18)    NOT NULL WITH DEFAULT,
   FUNCTION_TYPE    CHAR(2)     NOT NULL WITH DEFAULT,
   VIEW_CREATOR     CHAR(8)     NOT NULL WITH DEFAULT,
   VIEW_NAME        CHAR(18)    NOT NULL WITH DEFAULT,
   PATH             VARCHAR(254) NOT NULL WITH DEFAULT,
   FUNCTION_TEXT    VARCHAR(254) NOT NULL WITH DEFAULT);
```

Columns QUERYNO, QBLOCKNO, APPLNAME, PROGNAME, COLLID, and GROUP_MEMBER have the same meanings as in the PLAN_TABLE. See "Chapter 26. Using EXPLAIN to improve SQL performance" on page 667 for explanations of those columns. The meanings of the other columns are:

EXPLAIN_TIME
Timestamp when the EXPLAIN statement was executed.

SCHEMA_NAME
Schema name of the function that is invoked in the explained statement.

FUNCTION_NAME
Name of the function that is invoked in the explained statement.

SPEC_FUNC_NAME
Specific name of the function that is invoked in the explained statement.

FUNCTION_TYPE
The type of function that is invoked in the explained statement. Possible values are:

SU   Scalar function

TU   Table function

VIEW_CREATOR
The creator of the view, if the function that is specified in the FUNCTION_NAME column is referenced in a view definition. Otherwise, this field is blank.

VIEW_NAME
> The name of the view, if the function that is specified in the FUNCTION_NAME column is referenced in a view definition. Otherwise, this field is blank.

PATH
> The value of the SQL path when DB2 resolved the function reference.

FUNCTION_TEXT
> The text of the function reference (the function name and parameters). If the function reference exceeds 100 bytes, this column contains the first 100 bytes.
>
> For a function specified in infix notation, FUNCTION_TEXT contains only the function name. For example, suppose a user-defined function named / is in the function reference A/B. Then FUNCTION_TEXT contains only /, not A/B.

Casting of user-defined function arguments

Whenever you invoke a user-defined function, DB2 assigns your input parameter values to parameters with the data types and lengths in the user-defined function definition. See Chapter 2 of *DB2 SQL Reference* for information on how DB2 assigns values when the data types of the source and target differ.

When you invoke a user-defined function that is sourced on another function, DB2 casts your parameters to the data types and lengths of the sourced function.

The following example demonstrates what happens when the parameter definitions of a sourced function differ from those of the function on which it is sourced.

Suppose that external user-defined function TAXFN1 is defined like this:

```
CREATE FUNCTION TAXFN1(DEC(6,0))
  RETURNS DEC(5,2)
  PARAMETER STYLE DB2SQL
  LANGUAGE C
  EXTERNAL NAME TAXPROG;
```

Sourced user-defined function TAXFN2, which is sourced on TAXFN1, is defined like this:

```
CREATE FUNCTION TAXFN2(DEC(8,2))
  RETURNS DEC(5,0)
  SOURCE TAXFN1;
```

You invoke TAXFN2 using this SQL statement:

```
UPDATE TB1
  SET SALESTAX2 = TAXFN2(PRICE2);
```

TB1 is defined like this:

```
CREATE TABLE TB1
  (PRICE1    DEC(6,0),
   SALESTAX1 DEC(5,2),
   PRICE2    DEC(9,2),
   SALESTAX2 DEC(7,2));
```

Now suppose that PRICE2 has the DECIMAL(9,2) value 0001234.56. DB2 must first assign this value to the data type of the input parameter in the definition of TAXFN2, which is DECIMAL(8,2). The input parameter value then becomes 001234.56. Next, DB2 casts the parameter value to a source function parameter, which is DECIMAL(6,0). The parameter value then becomes 001234. (When you cast a value, that value is truncated, rather than rounded.)

Now, if TAXFN1 returns the DECIMAL(5,2) value 123.45, DB2 casts the value to DECIMAL(5,0), which is the result type for TAXFN2, and the value becomes 00123. This is the value that DB2 assigns to column SALESTAX2 in the UPDATE statement.

Casting of parameter markers: You can use untyped parameter markers in a function invocation. However, DB2 cannot compare the data types of untyped parameter markers to the data types of candidate functions. Therefore, DB2 might find more than one function that qualifies for invocation. If this happens, an SQL error occurs. To ensure that DB2 picks the right function to execute, cast the parameter markers in your function invocation to the data types of the parameters in the function that you want to execute. For example, suppose that two versions of function FX exist. One version of FX is defined with a parameter of type of DECIMAL(9,2), and the other is defined with a parameter of type INTEGER. You want to invoke FX with a parameter marker, and you want DB2 to execute the version of FX that has a DECIMAL(9,2) parameter. You need to cast the parameter marker to a DECIMAL(9,2) type:

```
SELECT FX(CAST(? AS DECIMAL(9,2))) FROM T1;
```

What happens when a user-defined function abnormally terminates

When an external user-defined function abnormally terminates, your program receives SQLCODE -430 for the invoking statement, and DB2 places the unit of work that contains the invoking statement in a must-rollback state. Include code in your program to check for a user-defined function abend and to roll back the unit of work that contains the user-defined function invocation.

Nesting SQL Statements

An SQL statement can explicitly invoke user-defined functions or stored procedures or can implicitly activate triggers that invoke user-defined functions or stored procedures. This is known as *nesting* of SQL statements. DB2 supports up to 16 levels of nesting. Figure 111 shows an example of SQL statement nesting.

```
Trigger TR1 is defined on table T3:
 CREATE TRIGGER TR1
  AFTER UPDATE ON T3
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
   CALL SP3(PARM1);
  END

Program P1 (nesting level 1) contains:
 SELECT UDF1(C1) FROM T1;

UDF1 (nesting level 2) contains:
 CALL SP2(C2);

SP2 (nesting level 3) contains:
 UPDATE T3 SET C3=1;

SP3 (nesting level 4) contains:
 SELECT UDF4(C4) FROM T4;
 .
 .
SP16 (nesting level 16) cannot invoke stored procedures
 or user-defined functions
```

Figure 111. Nested SQL statements

DB2 has the following restrictions on nested SQL statements:

- *Restrictions for SELECT statements:*

  When you execute a SELECT statement on a table, you cannot execute INSERT, UPDATE, or DELETE statements on the same table at a lower level of nesting.

  For example, suppose that you execute this SQL statement at level 1 of nesting:

  ```
  SELECT UDF1(C1) FROM T1;
  ```

  You cannot execute this SQL statement at a lower level of nesting:

  ```
  INSERT INTO T1 VALUES(...);
  ```

- *Restrictions for INSERT, UPDATE, and DELETE Statements:*

  When you execute an INSERT, DELETE, or UPDATE statement on a table, you cannot access that table from a user-defined function or stored procedure that is at a lower level of nesting.

  For example, suppose that you execute this SQL statement at level 1 of nesting:

  ```
  DELETE FROM T1 WHERE UDF3(T1.C1) = 3;
  ```

  You cannot execute this SELECT statement at a lower level of nesting:

  ```
  SELECT * FROM T1;
  ```

Although trigger activations count in the levels of SQL statement nesting, the previous restrictions on SQL statements do not apply to SQL statements that are executed in the trigger body. For example, suppose that trigger TR1 is defined on table T1:

```
CREATE TRIGGER TR1
 AFTER INSERT ON T1
 FOR EACH STATEMENT MODE DB2SQL
 BEGIN ATOMIC
  UPDATE T1 SET C1=1;
 END
```

Now suppose that you execute this SQL statement at level 1 of nesting:

```
   INSERT INTO T1 VALUES(...);
```

Although the UPDATE statement in the trigger body is at level 2 of nesting and modifies the same table that the triggering statement updates, DB2 can execute the INSERT statement successfully.

Recommendations for user-defined function invocation

Invoke user-defined functions with external actions and nondeterministic user-defined functions from select lists: It is better to invoke user-defined functions with external actions and nondeterministic user-defined functions from select lists, rather than predicates.

The access path that DB2 chooses for a predicate determines whether a user-defined function in that predicate is executed. To ensure that DB2 executes the external action for each row of the result set, put the user-defined function invocation in the SELECT list.

Invoking a nondeterministic user-defined function from a predicate can yield undesirable results. The following example demonstrates this idea.

Suppose that you execute this query:

```
SELECT COUNTER(), C1, C2 FROM T1 WHERE COUNTER() = 2;
```

Table T1 looks like this:

```
C1  C2
--  --
 1   b
 2   c
 3   a
```

COUNTER is a user-defined function that increments a variable in the scratchpad each time it is invoked.

DB2 invokes an instance of COUNTER in the predicate 3 times. Assume that COUNTER is invoked for row 1 first, for row 2 second, and for row 3 third. Then COUNTER returns 1 for row 1, 2 for row 2, and 3 for row 3. Therefore, row 2 satisfies the predicate WHERE COUNTER()=2, so DB2 evaluates the SELECT list for row 2. DB2 uses a different instance of COUNTER in the select list from the instance in the predicate. Because the instance of COUNTER in the select list is invoked only once, it returns a value of 1. Therefore, the result of the query is:

```
COUNTER()  C1 C2
---------  -- --
        1   2  c
```

This is not the result you might expect.

The results can differ even more, depending on the order in which DB2 retrieves the rows from the table. Suppose that an ascending index is defined on column C2. Then DB2 retrieves row 3 first, row 1 second, and row 2 third. This means that row 1 satisfies the predicate WHERE COUNTER()=2. The value of COUNTER in the select list is again 1, so the result of the query in this case is:

```
COUNTER()  C1 C2
---------  -- --
        1   1  b
```

*Understand the interaction between scrollable cursors and nondeterministic user-defined functions or user-defined functions with external actions:*When you use a scrollable cursor, you might retrieve the same row multiple times while the cursor is open. If the select list of the cursor's SELECT statement contains a user-defined function, that user-defined function is executed each time you retrieve a row. Therefore, if the user-defined function has an external action, and you retrieve the same row multiple times, the external action is executed multiple times for that row.

A similar situation occurs with scrollable cursors and nondeterministic functions. The result of a nondeterministic user-defined function can be different each time you execute the user-defined function. If the select list of a scrollable cursor contains a nondeterministic user-defined function, and you use that cursor to retrieve the same row multiple times, the results can differ each time you retrieve the row.

A nondeterministic user-defined function in the predicate of a scrollable cursor's SELECT statement does not change the result of the predicate while the cursor is open. DB2 evaluates a user-defined function in the predicate only once while the cursor is open.

Chapter 15. Creating and using distinct types

A distinct type is a data type that you define using the CREATE DISTINCT TYPE statement. Each distinct type has the same internal representation as a built-in data type. You can use distinct types in the same way that you use built-in data types, in any type of SQL application except for a DB2 private protocol application.

This chapter presents the following information about distinct types:
- "Introduction to distinct types"
- "Using distinct types in application programs" on page 300
- "Combining distinct types with user-defined functions and LOBs" on page 304

Introduction to distinct types

Suppose you want to define some audio and video data in a DB2 table. You can define columns for both types of data as BLOB, but you might want to use a data type that more specifically describes the data. To do that, define distinct types. You can then use those types when you define columns in a table or manipulate the data in those columns. For example, you can define distinct types for the audio and video data like this:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M);
CREATE DISTINCT TYPE VIDEO AS BLOB (1M);
```

Then, your CREATE TABLE statement might look like this:

```
CREATE TABLE VIDEO_CATALOG;
  (VIDEO_NUMBER  CHAR(6)  NOT NULL,
   VIDEO_SOUND   AUDIO,
   VIDEO_PICS    VIDEO,
   ROW_ID        ROWID NOT NULL GENERATED ALWAYS);
```

You must define a column of type ROWID in the table because tables with any type of LOB columns require a ROWID column, and internally, the VIDEO_CATALOG table contains two LOB columns. For more information on LOB data, see "Chapter 13. Programming for large objects (LOBs)" on page 229.

After you define distinct types and columns of those types, you can use those data types in the same way you use built-in types. You can use the data types in assignments, comparisons, function invocations, and stored procedure calls. However, when you assign one column value to another or compare two column values, those values must be of the same distinct type. For example, you must assign a column value of type VIDEO to a column of type VIDEO, and you can compare a column value of type AUDIO only to a column of type AUDIO. When you assign a host variable value to a column with a distinct type, you can use any host data type that is compatible with the source data type of the distinct type. For example, to receive an AUDIO or VIDEO value, you can define a host variable like this:

```
SQL TYPE IS BLOB (1M) HVAV;
```

When you use a distinct type as an argument to a function, a version of that function that accepts that distinct type must exist. For example, if function SIZE takes a BLOB type as input, you cannot automatically use a value of type AUDIO as input. However, you can create a sourced user-defined function that takes the AUDIO type as input. For example:

```
CREATE FUNCTION SIZE(AUDIO)
  RETURNS INTEGER
  SOURCE SIZE(BLOB(1M));
```

Using distinct types in application programs

The main reason to use distinct types is because DB2 enforces *strong typing* for distinct types. Strong typing ensures that only functions, procedures, comparisons, and assignments that are defined for a data type can be used.

For example, if you have defined a user-defined function to convert U.S. dollars to euro currency, you do not want anyone to use this same user-defined function to convert Japanese Yen to euros because the U.S. dollars to euros function returns the wrong amount. Suppose you define three distinct types:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL(9,2) WITH COMPARISONS;
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2) WITH COMPARISONS;
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2) WITH COMPARISONS;
```

If a conversion function is defined that takes an input parameter of type US_DOLLAR as input, DB2 returns an error if you try to execute the function with an input parameter of type JAPANESE_YEN.

Comparing distinct types

The basic rule for comparisons is that the data types of the operands must be compatible. The compatibility rule defines, for example, that all numeric types (SMALLINT, INTEGER, FLOAT, and DECIMAL) are compatible. That is, you can compare an INTEGER value with a value of type FLOAT. However, you cannot compare an object of a distinct type to an object of a different type. You can compare an object with a distinct type only to an object with exactly the same distinct type.

DB2 does not let you compare data of a distinct type directly to data of its source type. However, you can compare a distinct type to its source type by using a cast function.

For example, suppose you want to know which products sold more than US $100 000.00 in the US in the month of July in 1992 (7/92). Because you cannot compare data of type US_DOLLAR with instances of data of the source type of US_DOLLAR (DECIMAL) directly, you must use a cast function to cast data from DECIMAL to US_DOLLAR or from US_DOLLAR to DECIMAL. Whenever you create a distinct type, DB2 creates two cast functions, one to cast from the source type to the distinct type and the other to cast from the distinct type to the source type. For distinct type US_DOLLAR, DB2 creates a cast function called DECIMAL and a cast function called US_DOLLAR. When you compare an object of type US_DOLLAR to an object of type DECIMAL, you can use one of those cast functions to make the data types identical for the comparison. Suppose table US_SALES is defined like this:

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         US_DOLLAR);
```

Then you can cast DECIMAL data to US_DOLLAR like this:

```
SELECT PRODUCT_ITEM
  FROM    US_SALES
  WHERE   TOTAL > US_DOLLAR(100000.00)
  AND     MONTH = 7
  AND     YEAR  = 1992;
```

The casting satisfies the requirement that the compared data types are identical.

You cannot use host variables in statements that you prepare for dynamic execution. As explained in "Using parameter markers" on page 506, you can substitute parameter markers for host variables when you prepare a statement, and then use host variables when you execute the statement.

If you use a parameter marker in a predicate of a query, and the column to which you compare the value represented by the parameter marker is of a distinct type, you must cast the parameter marker to the distinct type, or cast the column to its source type.

For example, suppose distinct type CNUM is defined like this:

```
CREATE DISTINCT TYPE CNUM AS INTEGER WITH COMPARISONS;
```

Table CUSTOMER is defined like this:

```
CREATE TABLE CUSTOMER
  (CUST_NUM      CNUM NOT NULL,
   FIRST_NAME    CHAR(30) NOT NULL,
   LAST_NAME     CHAR(30) NOT NULL,
   PHONE_NUM     CHAR(20) WITH DEFAULT,
   PRIMARY KEY   (CUST_NUM));
```

In an application program, you prepare a SELECT statement that compares the CUST_NUM column to a parameter marker. Because CUST_NUM is of a distinct type, you must cast the distinct type to its source type:

```
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
  WHERE CAST(CUST_NUM AS INTEGER) = ?
```

Alternatively, you can cast the parameter marker to the distinct type:

```
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
  WHERE CUST_NUM = CAST (? AS CNUM)
```

Assigning distinct types

For assignments from columns to columns or from constants to columns of distinct types, the type of that value to be assigned must match the type of the object to which the value is assigned, or you must be able to cast one type to the other.

If you need to assign a value of one distinct type to a column of another distinct type, a function must exist that converts the value from one type to another. Because DB2 provides cast functions only between distinct types and their source types, you must write the function to convert from one distinct type to another.

Assigning column values to columns with different distinct types

Suppose tables JAPAN_SALES and JAPAN_SALES_98 are defined like this:

```
CREATE TABLE JAPAN_SALES
  (PRODUCT_ITEM   INTEGER,
   MONTH          INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR           INTEGER CHECK (YEAR > 1985),
   TOTAL          JAPANESE_YEN);
```

```
CREATE TABLE JAPAN_SALES_98
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR);
```

You need to insert values from the TOTAL column in JAPAN_SALES into the
TOTAL column of JAPAN_SALES_98. Because INSERT statements follow
assignment rules, DB2 does not let you insert the values directly from one column
to the other because the columns are of different distinct types. Suppose that a
user-defined function called US_DOLLAR has been written that accepts values of
type JAPANESE_YEN as input and returns values of type US_DOLLAR. You can
then use this function to insert values into the JAPAN_SALES_98 table:

```
INSERT INTO JAPAN_SALES_98
  SELECT PRODUCT_ITEM, US_DOLLAR(TOTAL)
  FROM JAPAN_SALES
  WHERE YEAR = 1998;
```

Assigning column values with distinct types to host variables
The rules for assigning distinct types to host variables or host variables to columns
of distinct types differ from the rules for constants and columns.

You can assign a column value of a distinct type to a host variable if you can assign
a column value of the distinct type's source type to the host variable. In the
following example, you can assign SIZECOL1 and SIZECOL2, which has distinct
type SIZE, to host variables of type double and short because the source type of
SIZE, which is INTEGER, can be assigned to host variables of type double or short.

```
EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);
:
SELECT SIZECOL1, SIZECOL2
  INTO :hv1, :hv2
  FROM TABLE1;
```

Assigning host variable values to columns with distinct types
When you assign a value in a host variable to a column with a distinct type, the
type of the host variable must be castable to the distinct type. For a table of base
data types and the base data types to which they can be cast, see Table 36 on
page 291.

In this example, values of host variable hv2 can be assigned to columns SIZECOL1
and SIZECOL2, because C data type short is equivalent to DB2 data type
SMALIINT, and SMALLINT is promotable to data type INTEGER. However, values
of hv1 cannot be assigned to SIZECOL1 and SIZECOL2, because C data type
double, which is equivalent to DB2 data type DOUBLE, is not promotable to data
type INTEGER.

```
EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);
:
INSERT INTO TABLE1
  VALUES (:hv1,:hv1);    /* Invalid statement */
INSERT INTO TABLE1
  VALUES (:hv2,:hv2);    /* Valid statement   */
```

Using distinct types in UNIONs

As with comparisons, DB2 enforces strong typing of distinct types in UNIONs. When you use a UNION to combine column values from several tables, the combined columns must be of the same types. For example, suppose you create a view that combines the values of the US_SALES, EUROPEAN_SALES, and JAPAN_SALES tables. The TOTAL columns in the three tables are of different distinct types, so before you combine the table values, you must convert the types of two of the TOTAL columns to the type of the third TOTAL column. Assume that the US_DOLLAR type has been chosen as the common distinct type. Because DB2 does not generate cast functions to convert from one distinct type to another, two user-defined functions must exist:

- A function that converts values of type EURO to US_DOLLAR
- A function that converts values of type JAPANESE_YEN to US_DOLLAR

Assume that these functions exist, and that both are called US_DOLLAR. Then you can execute a query like this to display a table of combined sales:

```
SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
FROM US_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR(TOTAL)
FROM EUROPEAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR(TOTAL)
FROM JAPAN_SALES;
```

Because the result type of both US_DOLLAR functions is US_DOLLAR, you have satisfied the requirement that the distinct types of the combined columns are the same.

Invoking functions with distinct types

DB2 enforces strong typing when you pass arguments to a function. This means that:

- You can pass arguments that have distinct types to a function if either of the following conditions is true:
  - A version of the function that accepts those distinct types is defined.

    This also applies to infix operators. If you want to use one of the five built-in infix operators (||, /, *, +, -) with your distinct types, you must define a version of that operator that accepts the distinct types.
  - You can cast your distinct types to the argument types of the function.
- If you pass arguments to a function that accepts only distinct types, the arguments you pass must have the same distinct types as in the function definition. If the types are different, you must cast your arguments to the distinct types in the function definition.

  If you pass constants or host variables to a function that accepts only distinct types, you must cast the constants or host variables to the distinct types that the function accepts.

The following examples demonstrate how to use distinct types as arguments in function invocations.

Example: Defining a function with distinct types as arguments: Suppose you want to invoke the built-in function HOUR with a distinct type that is defined like this:

```
CREATE DISTINCT TYPE FLIGHT_TIME AS TIME WITH COMPARISONS;
```

The HOUR function takes only the TIME or TIMESTAMP data type as an argument, so you need a sourced function that is based on the HOUR function that accepts the FLIGHT_TIME data type. You might declare a function like this:

```
CREATE FUNCTION HOUR(FLIGHT_TIME)
  RETURNS INTEGER
  SOURCE SYSIBM.HOUR(TIME);
```

Example: Casting function arguments to acceptable types: Another way you can invoke the HOUR function is to cast the argument of type FLIGHT_TIME to the TIME data type before you invoke the HOUR function. Suppose table FLIGHT_INFO contains column DEPARTURE_TIME, which has data type FLIGHT_TIME, and you want to use the HOUR function to extract the hour of departure from the departure time. You can cast DEPARTURE_TIME to the TIME data type, and then invoke the HOUR function:

```
SELECT HOUR(CAST(DEPARTURE_TIME AS TIME)) FROM FLIGHT_INFO;
```

Example: Using an infix operator with distinct type arguments: Suppose you want to add two values of type US_DOLLAR. Before you can do this, you must define a version of the + function that accepts values of type US_DOLLAR as operands:

```
CREATE FUNCTION "+"(US_DOLLAR,US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM."+"(DECIMAL(9,2),DECIMAL(9,2));
```

Because the US_DOLLAR type is based on the DECIMAL(9,2) type, the source function must be the version of + with arguments of type DECIMAL(9,2).

Example: Casting constants and host variables to distinct types to invoke a user-defined function: Suppose function CDN_TO_US is defined like this:

```
CREATE FUNCTION EURO_TO_US(EURO)
  RETURNS US_DOLLAR
  EXTERNAL NAME 'CDNCVT'
  PARAMETER STYLE DB2SQL
  LANGUAGE C;
```

This means that EURO_TO_US accepts only the EURO type as input. Therefore, if you want to call CDN_TO_US with a constant or host variable argument, you must cast that argument to distinct type EURO:

```
SELECT * FROM US_SALES
  WHERE TOTAL = EURO_TO_US(EURO(:H1));

SELECT * FROM US_SALES
  WHERE TOTAL = EURO_TO_US(EURO(10000));
```

Combining distinct types with user-defined functions and LOBs

The example in this section demonstrates the following concepts:
- Creating a distinct type based on a LOB data type
- Defining a user-defined function with a distinct type as an argument
- Creating a table with a distinct type column that is based on a LOB type
- Defining a LOB table space, auxiliary table, and auxiliary index
- Inserting data from a host variable into a distinct type column based on a LOB column
- Executing a query that contains a user-defined function invocation
- Casting a LOB locator to the input data type of a user-defined function

Suppose you keep electronic mail documents that are sent to your company in a DB2 table. The DB2 data type of an electronic mail document is a CLOB, but you

define it as a distinct type so that you can control the types of operations that are performed on the electronic mail. The distinct type is defined like this:

```
CREATE DISTINCT TYPE E_MAIL AS CLOB(5M);
```

You have also defined and written user-defined functions to search for and return the following information about an electronic mail document:
- Subject
- Sender
- Date sent
- Message content
- Indicator of whether the document contains a user-specified string

The user-defined function definitions look like this:

```
CREATE FUNCTION SUBJECT(E_MAIL)
  RETURNS VARCHAR(200)
  EXTERNAL NAME 'SUBJECT'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION SENDER(E_MAIL)
  RETURNS VARCHAR(200)
  EXTERNAL NAME 'SENDER'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION SENDING_DATE(E_MAIL)
  RETURNS DATE
  EXTERNAL NAME 'SENDDATE'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION CONTENTS(E_MAIL)
  RETURNS CLOB(1M)
  EXTERNAL NAME 'CONTENTS'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION CONTAINS(E_MAIL, VARCHAR (200))
  RETURNS INTEGER
  EXTERNAL NAME 'CONTAINS'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;
```

The table that contains the electronic mail documents is defined like this:

```
CREATE TABLE DOCUMENTS
  (LAST_UPDATE_TIME  TIMESTAMP,
   DOC_ROWID       ROWID NOT NULL GENERATED ALWAYS,
   A_DOCUMENT        E_MAIL);
```

Because the table contains a column with a source data type of CLOB, the table requires a ROWID column and an associated LOB table space, auxiliary table, and index on the auxiliary table. Use statements like this to define the LOB table space, the auxiliary table, and the index:

```
CREATE LOB TABLESPACE DOCTSLOB
  LOG YES
  GBPCACHE SYSTEM;

CREATE AUX TABLE DOCAUX_TABLE
  IN DOCTSLOB
  STORES DOCUMENTS COLUMN A_DOCUMENT;

CREATE INDEX A_IX_DOC ON DOCAUX_TABLE;
```

To populate the document table, you write code that executes an INSERT statement to put the first part of a document in the table, and then executes multiple UPDATE statements to concatenate the remaining parts of the document. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char hv_current_time[26];
  SQL TYPE IS CLOB (1M) hv_doc;
EXEC SQL END DECLARE SECTION;
/* Determine the current time and put this value */
/* into host variable hv_current_time.          */
/* Read up to 1 MB of document data from a file  */
/* into host variable hv_doc.                    */
  :
/* Insert the time value and the first 1 MB of   */
/* document data into the table.                 */
EXEC SQL INSERT INTO DOCUMENTS
  VALUES(:hv_current_time, DEFAULT, E_MAIL(:hv_doc));

/* While there is more document data in the      */
/* file, read up to 1 MB more of data, and then  */
/* use an UPDATE statement like this one to       */
/* concatenate the data in the host variable      */
/* to the existing data in the table.             */
EXEC SQL UPDATE DOCUMENTS
  SET A_DOCUMENT = A_DOCUMENT || E_MAIL(:hv_doc)
  WHERE LAST_UPDATE_TIME = :hv_current_time;
```

Now that the data is in the table, you can execute queries to learn more about the documents. For example, you can execute this query to determine which documents contain the word 'performance':

```
SELECT SENDER(A_DOCUMENT), SENDING_DATE(A_DOCUMENT),
  SUBJECT(A_DOCUMENT)
  FROM DOCUMENTS
  WHERE CONTAINS(A_DOCUMENT,'performance') = 1;
```

Because the electronic mail documents can be very large, you might want to use LOB locators to manipulate the document data instead of fetching all of a document into a host variable. You can use a LOB locator on any distinct type that is defined on one of the LOB types. The following example shows how you can cast a LOB locator as a distinct type, and then use the result in a user-defined function that takes a distinct type as an argument:

```
EXEC SQL BEGIN DECLARE SECTION
  long hv_len;
  char hv_subject[200];
  SQL TYPE IS CLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION
  :
/* Select a document into a CLOB locator.        */
```

```
EXEC SQL SELECT A_DOCUMENT, SUBJECT(A_DOCUMENT)
  INTO :hv_email_locator, :hv_subject
  FROM DOCUMENTS
  WHERE LAST_UPDATE_TIME = :hv_current_time;
  :
  :
/* Extract the subject from the document.  The    */
/* SUBJECT function takes an argument of type      */
/* E_MAIL, so cast the CLOB locator as E_MAIL.     */
EXEC SQL SET :hv_subject =
  SUBJECT(CAST(:hv_email_locator AS E_MAIL));
  :
  :
```

Part 4. Designing a DB2 database application

Chapter 16. Planning for DB2 program preparation

DB2 application programs include SQL statements. You need to process those SQL statements, using either the DB2 precompiler or an SQL statement coprocessor that is provided with a compiler. Either type of SQL statement processor does the following things:

- Replaces the SQL statements in your source programs with calls to DB2 language interface modules
- Creates a database request module (DBRM), which communicates your SQL requests to DB2 during the bind process

Figure 112 illustrates the program preparation process when you use the DB2 precompiler. Figure 113 on page 314 illustrates the program preparation process when you use an SQL statement coprocessor. "Chapter 20. Preparing an application program to run" on page 395 supplies specific details about accomplishing these steps.

After you have processed SQL statements in your source program, you create a load module, possibly one or more packages, and an application plan. Creating a load module involves compiling and link-editing the modified source code that is produced by the precompiler. Creating a package or an application plan, a process unique to DB2, involves binding one or more DBRMs.
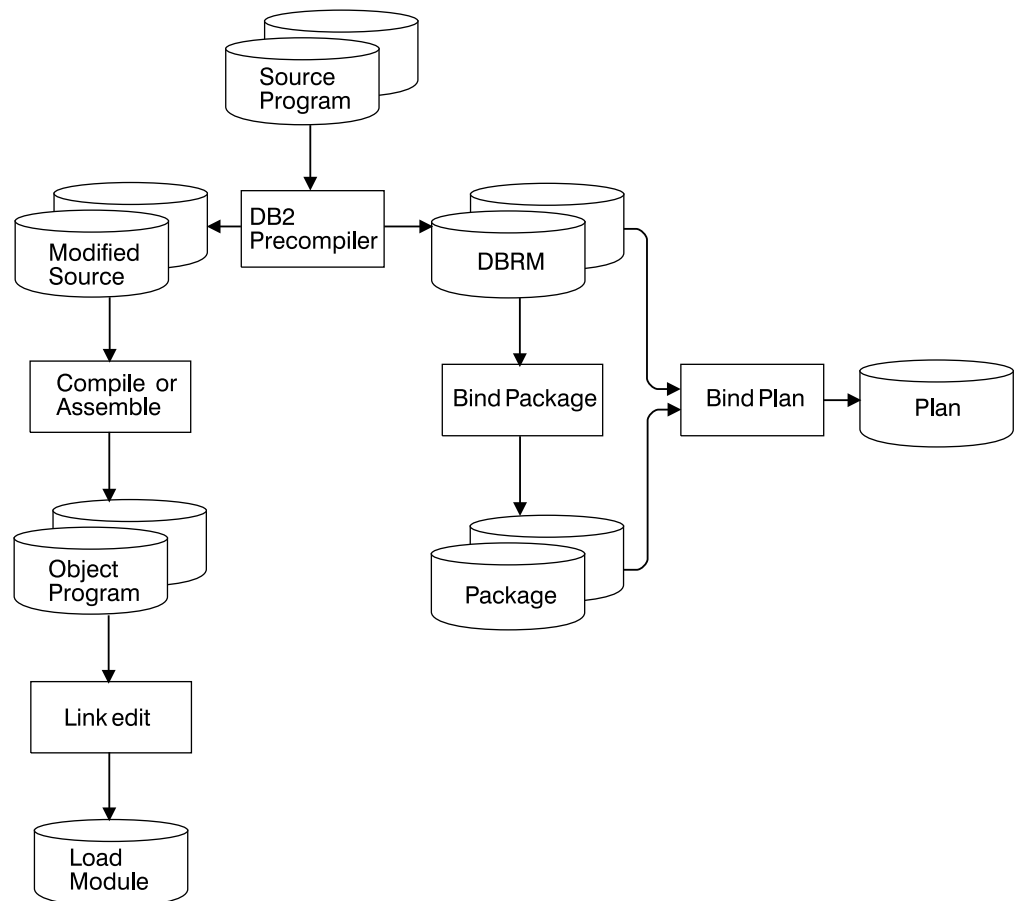
Figure 112. Program preparation with the DB2 precompiler
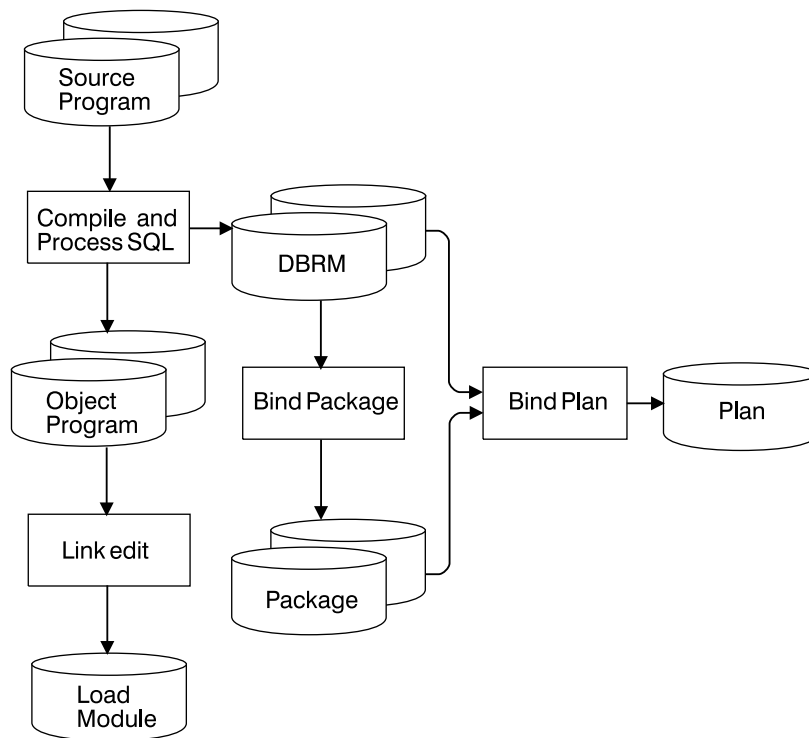
Figure 113. Program preparation with an SQL statement coprocessor

Planning to process SQL statements

When you process SQL statements in an application program, you can specify a number of options. Most of the options do not affect the way you design or code the program. Those options describe the basic characteristics of the source program or indicate how you want the output listings to look. For example, there are options that specify:

- The host language in which the program is written
- The maximum precision of decimal numbers in the program
- How many lines are on a page of the precompiler listing

In many cases, you may want to accept the default value provided.

A few options, however, can affect the way that you write your program. For example, you need to know if you are using NOFOR or STDSQL(YES) before you begin coding.

Before you begin writing your program, review the list of options in Table 48 on page 400. You can specify any of those options whether you use the DB2 precompiler or an SQL statement coprocessor. However, the SQL statement coprocessor might ignore certain options because there are compiler options that provide the same information.

Planning to bind

Depending upon how you design your DB2 application, you might bind all your DBRMs in one operation, creating only a single application plan. Or, you might bind some or all of your DBRMs into separate packages in separate operations. After that, you must still bind the entire application as a single plan, listing the included packages or collections and binding any DBRMs not already bound into packages. Regardless of what the plan contains, *you must bind a plan before the application can run*.

Binding or rebinding a package or plan in use: Packages and plans are locked when you bind or run them. Packages that run under a plan are not locked until the plan uses them. If you run a plan and some packages in the package list never run, those packages are never locked.

You cannot bind or rebind a package or a plan while it is running. However, you can bind a different version of a package that is running.

Options for binding and rebinding: Several of the options of BIND PACKAGE and BIND PLAN can affect your program design. For example, you can use a bind option to ensure that a package or plan can run only from a particular CICS connection or a particular IMS region—you do not have to enforce this in your code. Several other options are discussed at length in later chapters, particularly the ones that affect your program's use of locks, such as the option ISOLATION. Before you finish reading this chapter, you might want to review those options in Chapter 2 of *DB2 Command Reference*.

Preliminary steps: Before you bind, consider the following:
- Determine how you want to bind the DBRMs. You can bind them into packages, directly into plans, or use a combination of both methods.
- Develop a naming convention and strategy for the most effective and efficient use of your plans and packages.
- Determine when your application should acquire locks on the objects it uses: on all objects when the plan is first allocated, or on each object in turn when that object is first used. For a description of the consequences of either choice, see "The ACQUIRE and RELEASE options" on page 337.

Deciding how to bind DBRMs

The question of whether to use packages affects your application design from the beginning. For example, you might decide to put certain SQL statements together in the same program in order to precompile them into the same DBRM and then bind them into a single package.

Input to binding the plan can include DBRMs only, a package list only, or a combination of the two. When choosing one of those alternatives for your application, consider the impact of rebinding and see "Planning for changes to your application" on page 317.

Binding with a package list only
At one extreme, you can bind each DBRM into its own package. Input to binding a package is a single DBRM only. A one-to-one correspondence between programs and packages might easily allow you to keep track of each. However, your application could consist of too many packages to track easily.

Binding a plan that includes only a package list makes maintenance easier when the application changes significantly over time.

Binding all DBRMs to a plan

At the other extreme, you can bind all your DBRMs to a single plan. This approach has the disadvantage that a change to even one DBRM requires rebinding the entire plan, even though most DBRMs are unchanged.

Binding all DBRMs to a plan is suitable for small applications that are unlikely to change or that require all resources to be acquired when the plan is allocated rather than when your program first uses them.

Binding with both DBRMs and a package list

Binding DBRMs directly to the plan and specifying a package list is suitable for maintaining existing applications. You can add a package list when rebinding an existing plan. To migrate gradually to using packages, bind DBRMs as packages when you need to make changes.

Advantages of packages

You must decide how to use packages based on your application design and your operational objectives. Keep in mind the following:

Ease of maintenance: When you use packages, you do not need to bind the entire plan again when you change one SQL statement. You need to bind only the package associated with the changed SQL statement.

Incremental development of your program: Binding packages into package collections allows you to add packages to an existing application plan without having to bind the entire plan again. A *collection* is a group of associated packages. If you include a collection name in the package list when you bind a plan, any package in the collection becomes available to the plan. The collection can even be empty when you first bind the plan. Later, you can add packages to the collection, and drop or replace existing packages, without binding the plan again.

Versioning: Maintaining several versions of a plan without using packages requires a separate plan for each version, and therefore separate plan names and RUN commands. Isolating separate versions of a program into packages requires only one plan and helps to simplify program migration and fallback. For example, you can maintain separate development, test, and production levels of a program by binding each level of the program as a separate version of a package, all within a single plan.

Flexibility in using bind options: The options of BIND PLAN apply to all DBRMs bound directly to the plan. The options of BIND PACKAGE apply only to the single DBRM bound to that package. The package options need not all be the same as the plan options, and they need not be the same as the options for other packages used by the same plan.

Flexibility in using name qualifiers: You can use a bind option to name a qualifier for the unqualified object names in SQL statements in a plan or package. By using packages, you can use different qualifiers for SQL statements in different parts of your application. By rebinding, you can redirect your SQL statements, for example, from a test table to a production table.

> **CICS**
>
> With packages, you probably do not need dynamic plan selection and its accompanying exit routine. A package listed within a plan is not accessed until it is executed. However, it is possible to use dynamic plan selection and packages together. Doing so can reduce the number of plans in an application, and hence less effort to maintain the dynamic plan exit routine. See "Using packages with dynamic plan selection" on page 420 for information on using packages with dynamic plan selection.

Planning for changes to your application

As you design your application, consider what will happen to your plans and packages when you make changes to your application.

A change to your program probably invalidates one or more of your packages and perhaps your entire plan. For some changes, you must bind a new object; for others, rebinding is sufficient.

- To bind a new plan or package, other than a trigger package, use the subcommand BIND PLAN or BIND PACKAGE with the option ACTION(REPLACE).

  To bind a new trigger package, recreate the trigger associated with the trigger package.

- To rebind an existing plan or package, other than a trigger package, use the REBIND subcommand.

  To rebind trigger package, use the REBIND TRIGGER PACKAGE subcommand.

Table 37 tells which action particular types of change require. For more information on trigger packages, see "Working with trigger packages" on page 320.

If you want to change the bind options in effect when the plan or package runs, review the descriptions of those options in Chapter 2 of *DB2 Command Reference*. Not all options of BIND are also available on REBIND.

A plan or package can also become invalid for reasons that do not depend on operations in your program: for example, if an index is dropped that is used as an access path by one of your queries. In those cases, DB2 might rebind the plan or package automatically, the next time it is used. (For details about that operation, see "Automatic rebinding" on page 320.)

Table 37. Changes requiring BIND or REBIND

Change made:	Minimum action necessary:
Drop a table, index or other object, and recreate the object	If a table with a trigger is dropped, recreate the trigger if you recreate the table. Otherwise, no change is required; automatic rebind is attempted at the next run.
Revoke an authorization to use an object	None required; automatic rebind is attempted at the next run. Automatic rebind fails if authorization is still not available; then you must issue REBIND for the package or plan.
Run RUNSTATS to update catalog statistics	Issue REBIND for the package or plan to possibly change the access path chosen.

Table 37. Changes requiring BIND or REBIND (continued)

Change made:	Minimum action necessary:
Add an index to a table	Issue REBIND for the package or plan to use the index.
Change bind options	Issue REBIND for the package or plan, or issue BIND with ACTION(REPLACE) if the option you want is not available on REBIND.
Change statements in host language *and* SQL statements	Precompile, compile, and link the application program. Issue BIND with ACTION(REPLACE) for the package or plan.

Dropping objects

If you drop an object that a package depends on, the following occurs:

- If the package is not appended to any running plan, the package becomes invalid.
- If the package is appended to a running plan, and the drop occurs outside of that plan, the object is not dropped, and the package does not become invalid.
- If the package is appended to a running plan, and the drop occurs within that plan, the package becomes invalid.

In all cases, the plan does not become invalid unless it has a DBRM referencing the dropped object. If the package or plan becomes invalid, automatic rebind occurs the next time the package or plan is allocated.

Rebinding a package

Table 38 clarifies which packages are bound, depending on how you specify *collection-id* (coll-id) *package-id* (pkg-id), and *version-id* (ver-id) on the REBIND PACKAGE subcommand. For syntax and descriptions of this subcommand, see Chapter 2 of *DB2 Command Reference*.

REBIND PACKAGE does not apply to packages for which you do not have the BIND privilege. An asterisk (*) used as an identifier for collections, packages, or versions does not apply to packages at remote sites.

Table 38. Behavior of REBIND PACKAGE specification. "All" means all collections, packages, or versions at the local DB2 server for which the authorization ID that issues the command has the BIND privilege. The symbol '•.' stands for a required period in the command syntax; '★' stands for an asterisk.

INPUT	Collections Affected	Packages Affected	Versions Affected
★	all	all	all
★•★•(★)	all	all	all
★•★	all	all	all
★•★•(ver-id)	all	all	ver-id
★•★•()	all	all	empty string
coll-id•★	coll-id	all	all
coll-id•★•(★)	coll-id	all	all
coll-id•★•(ver-id)	coll-id	all	ver-id
coll-id•★•()	coll-id	all	empty string
coll-id•pkg-id•(★)	coll-id	pkg-id	all

Table 38. Behavior of REBIND PACKAGE specification (continued). "All" means all collections, packages, or versions at the local DB2 server for which the authorization ID that issues the command has the BIND privilege. The symbol '. ' stands for a required period in the command syntax; '★' stands for an asterisk.

INPUT	Collections Affected	Packages Affected	Versions Affected
coll-id•pkg-id	coll-id	pkg-id	empty string
coll-id•pkg-id•()	coll-id	pkg-id	empty string
coll-id•pkg-id•(ver-id)	coll-id	pkg-id	ver-id
★•pkg-id•(★)	all	pkg-id	all
★•pkg-id	all	pkg-id	empty string
★•pkg-id•()	all	pkg-id	empty string
★•pkg-id•(ver-id)	all	pkg-id	ver-id

The following example shows the options for rebinding a package at the remote location, SNTERSA. The collection is GROUP1, the package ID is PROGA, and the version ID is V1. The connection types shown in the REBIND subcommand replace connection types specified on the original BIND subcommand. For information on the REBIND subcommand options, see *DB2 Command Reference*.

```
REBIND PACKAGE(SNTERSA.GROUP1.PROGA.(V1)) ENABLE(CICS,REMOTE)
```

You can use the asterisk on the REBIND subcommand for local packages, but not for packages at remote sites. Any of the following commands rebinds all versions of all packages in all collections, at the local DB2 system, for which you have the BIND privilege.

REBIND PACKAGE (*)
REBIND PACKAGE (*.*)
REBIND PACKAGE (*.*.(*))

Either of the following commands rebinds all versions of all packages in the local collection LEDGER for which you have the BIND privilege.

REBIND PACKAGE (LEDGER.*)
REBIND PACKAGE (LEDGER.*.(*))

Either of the following commands rebinds the empty string version of the package DEBIT in all collections, at the local DB2 system, for which you have the BIND privilege.

REBIND PACKAGE (*.DEBIT)
REBIND PACKAGE (*.DEBIT.())

Rebinding a plan

Using the PKLIST keyword replaces any previously specified package list. Omitting the PKLIST keyword allows the use of the previous package list for rebinding. Using the NOPKLIST keyword deletes any package list specified when the plan was previously bound.

The following example rebinds PLANA and changes the package list.

REBIND PLAN(PLANA) PKLIST(GROUP1.*) MEMBER(ABC)

The following example rebinds the plan and drops the entire package list.

REBIND PLAN(PLANA) NOPKLIST

Rebinding lists of plans and packages

You can generate a list of REBIND subcommands for a set of plans or packages that cannot be described by using asterisks, using information in the DB2 catalog. You can then issue the list of subcommands through DSN.

One situation in which the technique is particularly useful is in completing a rebind operation that has terminated for lack of resources. A rebind for many objects, say REBIND PACKAGE (*) for an ID with SYSADM authority, terminates if a needed resource becomes unavailable. As a result, some objects are successfully rebound and others are not. If you repeat the subcommand, DB2 attempts to rebind all the objects again. But if you generate a rebind subcommand for each object that was not rebound, and issue those, DB2 does not repeat any work already done and is not likely to run out of resources.

For a description of the technique and several examples of its use, see "Appendix E. REBIND subcommands for lists of plans or packages" on page 911.

Working with trigger packages

A trigger package is a special type of package that is created only when you execute a CREATE TRIGGER statement. A trigger package executes only when the trigger with which it is associated is activated.

As with any other package, DB2 marks a trigger package invalid when you drop a table, index, or view on which the trigger package depends. DB2 executes an automatic rebind the next time the trigger activates. However, if the automatic rebind fails, DB2 does not mark the trigger package inoperative.

Unlike other packages, a trigger package is freed if you drop the table on which the trigger is defined, so you can recreate the trigger package only by recreating the table and the trigger.

You can use the subcommand REBIND TRIGGER PACKAGE to rebind a trigger package that DB2 has marked inoperative. You can also use REBIND TRIGGER PACKAGE to change the option values with which DB2 originally bound the trigger package. The default values for the options that you can change are:
• CURRENTDATA(YES)
• EXPLAIN(YES)
• FLAG(I)
• ISOLATION(RR)
• IMMEDWRITE(NO)
• RELEASE(COMMIT)

When you run REBIND TRIGGER PACKAGE, you can change only the values of options CURRENTDATA, EXPLAIN, FLAG, IMMEDWRITE, ISOLATION, and RELEASE.

Automatic rebinding

Automatic rebind might occur if an authorized user invokes a plan or package when the attributes of the data on which the plan or package depends change, or if the environment in which the package executes changes. Whether the automatic rebind occurs depends on the value of the field AUTO BIND on installation panel DSNTIPO. The options used for an automatic rebind are the options used during the most recent bind process.

In most cases, DB2 marks a plan or package that needs to be automatically rebound as *invalid*. A few common situations in which DB2 marks a plan or package as invalid are:

- When a table, index, or view on which the plan or package depends is dropped
- When the authorization of the owner to access any of those objects is revoked
- When the authorization to execute a stored procedure is revoked from a plan or package owner, and the plan or package uses the CALL *literal* form to call the stored procedure
- When a table on which the plan or package depends is altered to add a TIME, TIMESTAMP, or DATE column
- When a created temporary table on which the plan or package depends is altered to add a column
- When a user-defined function on which the plan or package depends is altered
- When a table is altered to add a self-referencing constraint or a constraint with a delete rule of SET NULL or CASCADE

Whether a plan or package is valid is recorded in column VALID of catalog tables SYSPLAN and SYSPACKAGE.

In the following cases, DB2 might automatically rebind a plan or package that has not been marked as invalid:

- A plan or package is bound in a different release of DB2 from the release in which it was first used.
- A plan or package has a location dependency and runs at a location other than the one at which it was bound. This can happen when members of a data sharing group are defined with location names, and a package runs on a different member from the one on which it was bound.

DB2 marks a plan or package as *inoperative* if an automatic rebind fails. Whether a plan or package is operative is recorded in column OPERATIVE of SYSPLAN and SYSPACKAGE.

Whether EXPLAIN runs during automatic rebind depends on the value of the field EXPLAIN PROCESSING on installation panel DSNTIPO, and on whether you specified EXPLAIN(YES). Automatic rebind fails for all EXPLAIN errors except "PLAN_TABLE not found."

The SQLCA is not available during automatic rebind. Therefore, if you encounter lock contention during an automatic rebind, DSNT501I messages cannot accompany any DSNT376I messages that you receive. To see the matching DSNT501I messages, you must issue the subcommand REBIND PLAN or REBIND PACKAGE.

Chapter 17. Planning for concurrency

This chapter begins with an overview of concurrency and locks in the following sections:
* "Definitions of concurrency and locks",
* "Effects of DB2 locks" on page 324, and
* "Basic recommendations to promote concurrency" on page 327.

After the basic recommendations, the chapter tells what you can do about a major technique that DB2 uses to control concurrency.
* **Transaction locks** mainly control access by SQL statements. Those locks are the ones over which you have the most control.
  - "Aspects of transaction locks" on page 331 describes the various types of transaction locks that DB2 uses and how they interact.
  - "Lock tuning" on page 337 describes what you can change to control locking. Your choices include:
    - "Bind options" on page 337
    - "Isolation overriding with SQL statements" on page 349
    - "The statement LOCK TABLE" on page 350

  Under those headings, *lock* (with no qualifier) refers to *transaction lock*.

Two other techniques also control concurrency in some situations.
* **Claims and drains** control access by DB2 utilities and commands. For information about them, see Part 5 (Volume 2) of *DB2 Administration Guide*.
* **Physical locks** are of concern only if you are using DB2 data sharing. For information about that, see *DB2 Data Sharing: Planning and Administration*.

The final section of this chapter describes locking activity for LOBs. See "LOB locks" on page 353.

Definitions of concurrency and locks

Definition: *Concurrency* is the ability of more than one application process to access the same data at essentially the same time.

Example: An application for order entry is used by many transactions simultaneously. Each transaction makes inserts in tables of invoices and invoice items, reads a table of data about customers, and reads and updates data about items on hand. Two operations on the same data, by two simultaneous transactions, might be separated only by microseconds. To the users, the operations appear concurrent.

Conceptual background: Concurrency must be controlled to prevent lost updates and such possibly undesirable effects as unrepeatable reads and access to uncommitted data.

  Lost updates. Without concurrency control, two processes, A and B, might both read the same row from the database, and both calculate new values for one of its columns, based on what they read. If A updates the row with its new value, and then B updates the same row, A's update is lost.

  Access to uncommitted data. Also without concurrency control, process A might update a value in the database, and process B might read that value

before it was committed. Then, if A's value is not later committed, but backed out, B's calculations are based on uncommitted (and presumably incorrect) data.

Unrepeatable reads. Some processes require the following sequence of events: A reads a row from the database and then goes on to process other SQL requests. Later, A reads the first row again and must find the same values it read the first time. Without control, process B could have changed the row between the two read operations.

To prevent those situations from occurring unless they are specifically allowed, DB2 might use *locks* to control concurrency.

What do locks do? A lock associates a DB2 resource with an application process in a way that affects how other processes can access the same resource. The process associated with the resource is said to "hold" or "own" the lock. DB2 uses locks to ensure that no process accesses data that has been changed, but not yet committed, by another process.

What do you do about locks? To preserve data integrity, your application process acquires locks implicitly, that is, under DB2 control. It is not necessary for a process to request a lock explicitly to conceal uncommitted data. Therefore, sometimes you need not do anything about DB2 locks. Nevertheless processes acquire, or avoid acquiring, locks based on certain general parameters. You can make better use of your resources and improve concurrency by understanding the effects of those parameters.

Effects of DB2 locks

The effects of locks that you want to minimize are *suspension*, *timeout*, and *deadlock*.

Suspension

Definition: An application process is *suspended* when it requests a lock that is already held by another application process and cannot be shared. The suspended process temporarily stops running.

Order of precedence for lock requests: Incoming lock requests are queued. Requests for lock promotion, and requests for a lock by an application process that already holds a lock on the same object, precede requests for locks by new applications. Within those groups, the request order is "first in, first out".

Example: Using an application for inventory control, two users attempt to reduce the quantity on hand of the same item at the same time. The two lock requests are queued. The second request in the queue is suspended and waits until the first request releases its lock.

Effects: The suspended process resumes running when:
- All processes that hold the conflicting lock release it.
- The requesting process times out or deadlocks and the process resumes to deal with an error condition.

Timeout

Definition: An application process is said to *time out* when it is terminated because it has been suspended for longer than a preset interval.

Example: An application process attempts to update a large table space that is being reorganized by the utility REORG TABLESPACE with SHRLEVEL NONE. It is likely that the utility job will not release control of the table space before the application process times out.

Effects: DB2 terminates the process, issues two messages to the console, and returns SQLCODE -911 or -913 to the process (SQLSTATEs '40001' or '57033'). Reason code 00C9008E is returned in the SQLERRD(3) field of the SQLCA. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0196.

IMS

If you are using IMS, and a timeout occurs, the following actions take place:
- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
  - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
  - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
  - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

COMMIT and ROLLBACK operations do not time out. The command STOP DATABASE, however, may time out and send messages to the console, but it will retry up to 15 times.

Deadlock

Definition: A *deadlock* occurs when two or more application processes each hold locks on resources that the others need and without which they cannot proceed.

Example: Figure 114 on page 326 illustrates a deadlock between two transactions.
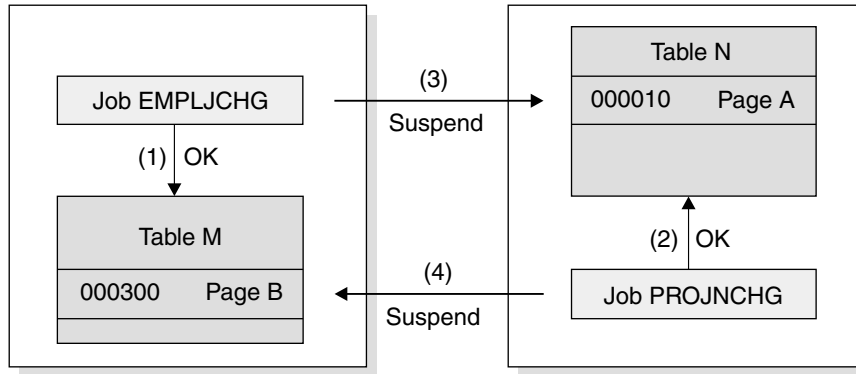
Notes:

1. Jobs EMPLJCHG and PROJNCHG are two transactions. Job EMPLJCHG accesses table M, and acquires an exclusive lock for page B, which contains record 000300.
2. Job PROJNCHG accesses table N, and acquires an exclusive lock for page A, which contains record 000010.
3. Job EMPLJCHG requests a lock for page A of table N while still holding the lock on page B of table M. The job is suspended, because job PROJNCHG is holding an exclusive lock on page A.
4. Job PROJNCHG requests a lock for page B of table M while still holding the lock on page A of table N. The job is suspended, because job EMPLJCHG is holding an exclusive lock on page B. The situation is a deadlock.

Figure 114. A deadlock example

Effects: After a preset time interval (the value of DEADLOCK TIME), DB2 can roll back the current unit of work for one of the processes or request a process to terminate. That frees the locks and allows the remaining processes to continue. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0172. Reason code 00C90088 is returned in the SQLERRD(3) field of the SQLCA.

It is possible for two processes to be running on distributed DB2 subsystems, each trying to access a resource at the other location. In that case, neither subsystem can detect that the two processes are in deadlock; the situation resolves only when one process times out.

Indications of deadlocks: In some cases, a deadlock can occur if two application processes attempt to update data in the same page or table space.

TSO, Batch, and CAF

When a deadlock or timeout occurs in these environments, DB2 attempts to roll back the SQL for one of the application processes. If the ROLLBACK is successful, that application receives SQLCODE -911. If the ROLLBACK fails, and the application does not abend, the application receives SQLCODE -913.

If you are using IMS, and a deadlock occurs, the following actions take place:
- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
  - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
  - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
  - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

If you are using CICS and a deadlock occurs, the CICS attachment facility decides whether or not to roll back one of the application processes, based on the value of the ROLBE or ROLBI parameter. If your application process is chosen for rollback, it receives one of two SQLCODEs in the SQLCA:

-911 A SYNCPOINT command with the ROLLBACK option was issued on behalf of your application process. All updates (CICS commands and DL/I calls, as well as SQL statements) that occurred during the current unit of work have been undone. (SQLSTATE '40001')

-913 A SYNCPOINT command with the ROLLBACK option was not issued. DB2 rolls back only the incomplete SQL statement that encountered the deadlock or timed out. CICS does not roll back any resources. Your application process should either issue a SYNCPOINT command with the ROLLBACK option itself or terminate. (SQLSTATE '57033')

Consider using the DSNTIAC subroutine to check the SQLCODE and display the SQLCA. Your application must take appropriate actions before resuming.

Basic recommendations to promote concurrency

Recommendations are grouped roughly by their scope, as:
- "Recommendations for database design" on page 328
- "Recommendations for application design" on page 328

Recommendations for database design

Keep like things together: Cluster tables relevant to the same application into the same database, and give each application process that creates private tables a private database in which to do it. In the ideal model, each application process uses as few databases as possible.

Keep unlike things apart: Give users different authorization IDs for work with different databases; for example, one ID for work with a shared database and another for work with a private database. This effectively adds to the number of possible (but not concurrent) application processes while minimizing the number of databases each application process can access.

Plan for batch inserts: If your application does sequential batch insertions, excessive contention on the space map pages for the table space can occur. This problem is especially apparent in data sharing, where contention on the space map means the added overhead of page P-lock negotiation. For these types of applications, consider using the MEMBER CLUSTER option of CREATE TABLESPACE. This option causes DB2 to disregard the clustering index (or implicit clustering index) when assigning space for the SQL INSERT statement. For more information about using this option in data sharing, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*. For the syntax, see Chapter 5 of *DB2 SQL Reference*.

Use LOCKSIZE ANY until you have reason not to: LOCKSIZE ANY is the default for CREATE TABLESPACE. It allows DB2 to choose the lock size, and DB2 usually chooses LOCKSIZE PAGE and LOCKMAX SYSTEM for non-LOB table spaces. For LOB table spaces, it chooses LOCKSIZE LOB and LOCKMAX SYSTEM. You should use LOCKSIZE TABLESPACE or LOCKSIZE TABLE only for read-only table spaces or tables, or when concurrent access to the object is not needed. Before you choose LOCKSIZE ROW, you should estimate whether there will be an increase in overhead for locking and weigh that against the increase in concurrency.

Examine small tables: For small tables with high concurrency requirements, estimate the number of pages in the data and in the index. If the index entries are short or they have many duplicates, then the entire index can be one root page and a few leaf pages. In this case, spread out your data to improve concurrency, or consider it a reason to use row locks.

Partition the data: Online queries typically make few data changes, but they occur often. Batch jobs are just the opposite; they run for a long time and change many rows, but occur infrequently. The two do not run well together. You might be able to separate online applications from batch, or two batch jobs from each other. To separate online and batch applications, provide separate partitions. Partitioning can also effectively separate batch jobs from each other.

Fewer rows of data per page: By using the MAXROWS clause of CREATE or ALTER TABLESPACE, you can specify the maximum number of rows that can be on a page. For example, if you use MAXROWS 1, each row occupies a whole page, and you confine a page lock to a single row. Consider this option if you have a reason to avoid using row locking, such as in a data sharing environment where row locking overhead can be excessive.

Recommendations for application design

Access data in a consistent order: When different applications access the same data, try to make them do so in the same sequence. For example, make both

access rows 1,2,3,5 in that order. In that case, the first application to access the data delays the second, but the two applications cannot deadlock. For the same reason, try to make different applications access the same tables in the same order.

Commit work as soon as is practical: To avoid unnecessary lock contentions, issue a COMMIT statement as soon as possible after reaching a point of consistency, even in read-only applications. To prevent unsuccessful SQL statements (such as PREPARE) from holding locks, issue a ROLLBACK statement after a failure. Statements issued through SPUFI can be committed immediately by the SPUFI autocommit feature.

Taking commit points frequently in a long running unit of recovery (UR) has the following benefits:
- Reduces lock contention
- Improves the effectiveness of lock avoidance, especially in a data sharing environment
- Reduces the elapsed time for DB2 system restart following a system failure
- Reduces the elapsed time for a unit of recovery to rollback following an application failure or an explicit rollback request by the application
- Provides more opportunity for utilities, such as online REORG, to break in

Consider using the UR CHECK FREQ field or the UR LOG WRITE CHECK field of installation panel DSNTIPN to help you identify those applications that are not committing frequently. UR CHECK FREQ, which identifies when too many checkpoints have occurred without a UR issuing a commit, is helpful in monitoring overall system activity. UR LOG WRITE CHECK enables you to detect applications that might write too many log records between commit points, potentially creating a lengthy recovery situation for critical tables.

Even though an application might conform to the commit frequency standards of the installation under normal operational conditions, variation can occur based on system workload fluctuations. For example, a low-priority application might issue a commit frequently on a system that is lightly loaded. However, under a heavy system load, the use of the CPU by the application may be pre-empted, and, as a result, the application may violate the rule set by the UR CHECK FREQ parameter. For this reason, add logic to your application to commit based on time elapsed since last commit, and not solely based on the amount of SQL processing performed. In addition, take frequent commit points in a long running unit of work that is read-only to reduce lock contention and to provide opportunities for utilities, such as online REORG, to access the data.

Retry an application after deadlock or timeout: Include logic in a batch program so that it retries an operation after a deadlock or timeout. Such a method could help you recover from the situation without assistance from operations personnel. Field SQLERRD(3) in the SQLCA returns a reason code that indicates whether a deadlock or timeout occurred.

Close cursors: If you define a cursor using the WITH HOLD option, the locks it needs can be held past a commit point. Use the CLOSE CURSOR statement as soon as possible in your program to cause those locks to be released and the resources they hold to be freed at the first commit point that follows the CLOSE CURSOR statement. Whether page or row locks are held for WITH HOLD cursors is controlled by the RELEASE LOCKS parameter on panel DSNTIP4.

Free locators: If you have executed, the HOLD LOCATOR statement, the LOB locator holds locks on LOBs past commit points. Use the FREE LOCATOR statement to release these locks.

Bind plans with ACQUIRE(USE): ACQUIRE(USE), which indicates that DB2 will acquire table and table space locks when the objects are first used and not when the plan is allocated, is the best choice for concurrency. Packages are always bound with ACQUIRE(USE), by default. ACQUIRE(ALLOCATE) can provide better protection against timeouts. Consider ACQUIRE(ALLOCATE) for applications that need gross locks instead of intent locks or that run with other applications that may request gross locks instead of intent locks. Acquiring the locks at plan allocation also prevents any one transaction in the application from incurring the cost of acquiring the table and table space locks. If you need ACQUIRE(ALLOCATE), you might want to bind all DBRMs directly to the plan.

Bind with ISOLATION(CS) and CURRENTDATA(NO) typically: ISOLATION(CS) lets DB2 release acquired row and page locks as soon as possible. CURRENTDATA(NO) lets DB2 avoid acquiring row and page locks as often as possible. After that, in order of decreasing preference for concurrency, use these bind options:

1. ISOLATION(CS) with CURRENTDATA(YES), when data returned to the application must not be changed before your next FETCH operation.

2. ISOLATION(RS), when data returned to the application must not be changed before your application commits or rolls back. However, you do not care if other application processes insert additional rows.

3. ISOLATION(RR), when data evaluated as the result of a query must not be changed before your application commits or rolls back. New rows cannot be inserted into the answer set.

For updatable scrollable cursors, ISOLATION(CS) provides the additional advantage of letting DB2 use *optimistic concurrency control* to further reduce the amount of time that locks are held. For more information about optimistic concurrency control, see "Advantages and disadvantages of the isolation values" on page 341.

Use ISOLATION(UR) cautiously: UR isolation acquires almost no locks on rows or pages. It is fast and causes little contention, but it reads uncommitted data. Do not use it unless you are sure that your applications and end users can accept the logical inconsistencies that can occur.

Use global transactions: The Recoverable Resource Manager Services attachment facility (RRSAF) relies on an OS/390 component called OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS). OS/390 RRS provides system-wide services for coordinating two-phase commit operations across MVS products. For RRSAF applications and IMS transactions that run under OS/390 RRS, you can group together a number of DB2 agents into a single global transaction. A global transaction allows multiple DB2 agents to participate in a single global transaction and thus share the same locks and access the same data. When two agents that are in a global transaction access the same DB2 object within a unit of work, those agents will not deadlock with each other. The following restrictions apply:

• There is no Parallel Sysplex support for global transactions.

• Because each of the "branches" of a global transaction are sharing locks, uncommitted updates issued by one branch of the transaction are visible to other branches of the transaction.

- Claim/drain processing is not supported across the branches of a global transaction, which means that attempts to issue CREATE, DROP, ALTER, GRANT, or REVOKE may deadlock or timeout if they are requested from different branches of the same global transaction.
- Attempts to update a partitioning key may deadlock or timeout because of the same restrictions on claim/drain processing.
- LOCK TABLE may deadlock or timeout across the branches of a global transaction.

For information on how to make an agent part of a global transaction for RRSAF applications, see "Chapter 30. Programming for the Recoverable Resource Manager Services attachment facility (RRSAF)" on page 763.

Aspects of transaction locks

Transaction locks have the following four basic aspects:
- "The size of a lock"
- "The duration of a lock" on page 333
- "The mode of a lock" on page 334
- "The object of a lock" on page 336

Knowing the aspects helps you understand why a process suspends or times out or why two processes deadlock.

The size of a lock

Definition
The *size* (sometimes *scope* or *level*) of a lock on data in a table describes the amount of data controlled. The possible sizes of locks are table space, table, partition, page, and row. This section contains information about locking for non-LOB data. See "LOB locks" on page 353 for information on locking for LOBs.

Hierarchy of lock sizes
The same piece of data can be controlled by locks of different sizes. A table space lock (the largest size) controls the most data, all the data in an entire table space. A page or row lock controls only the data in a single page or row.

As Figure 115 on page 332 suggests, row locks and page locks occupy an equal place in the hierarchy of lock sizes.
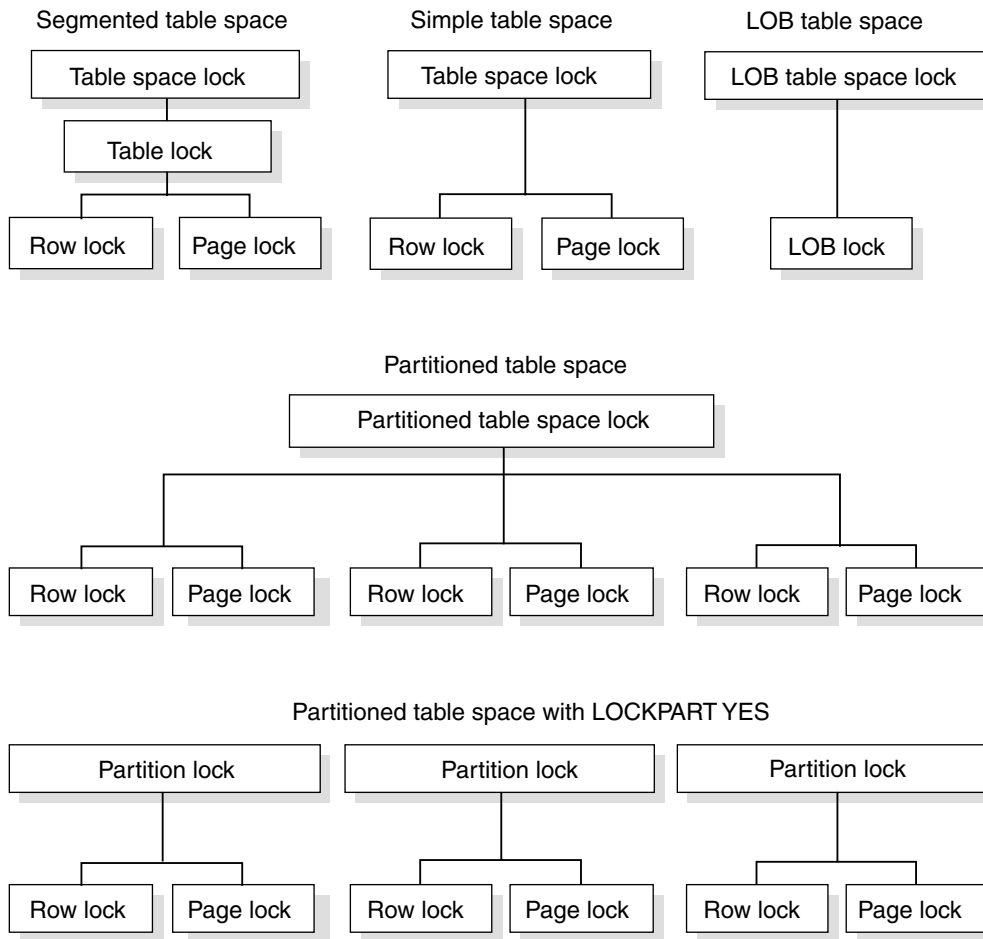
Figure 115. Sizes of objects locked

General effects of size

Locking larger or smaller amounts of data allows you to trade performance for concurrency. Using page or row locks instead of table or table space locks has the following effects:

- Concurrency usually improves, meaning better response times and higher throughput rates for many users.
- Processing time and use of storage increases. That is especially evident in batch processes that scan or update a large number of rows.

Using only table or table space locks has the following effects:

- Processing time and storage usage is reduced.
- Concurrency can be reduced, meaning longer response times for some users but better throughput for one user.

Effects of table spaces of different types

- The LOCKPART clause of CREATE and ALTER TABLESPACE lets you control how DB2 locks **partitioned table spaces**. The default, LOCKPART NO, means that one lock is used to lock the entire partitioned table space when any partition is accessed. LOCKPART NO is the value you want in most cases.

  With LOCKPART YES, individual partitions are locked only as they are accessed.

One case for using LOCKPART YES is for some data sharing applications, as described in Chapter 6 of *DB2 Data Sharing: Planning and Administration*. There are also benefits to non-data-sharing applications that use partitioned table spaces. For these applications, it might be desirable to acquire gross locks (S, U, or X) on partitions to avoid numerous lower level locks and yet still maintain concurrency. When locks escalate and the table space is defined with LOCKPART YES, applications that access different partitions of the same table space do not conflict during update activity.

Restrictions: If any of the following conditions are true, DB2 must lock *all* partitions when LOCKPART YES is used:
– The plan is bound with ACQUIRE(ALLOCATE).
– The table space is defined with LOCKSIZE TABLESPACE.
– LOCK TABLE IN EXCLUSIVE MODE or LOCK TABLE IN SHARE MODE is used (without the PART option).

No matter how LOCKPART is defined, utility jobs can control separate partitions of a table space or index space and can run concurrently with operations on other partitions.

- A **simple table space** can contain more than one table. A lock on the table space locks all the data in every table. A single page of the table space can contain rows from every table. A lock on a page locks every row in the page, no matter what tables the data belongs to. Thus, a lock needed to access data from one table can make data from other tables temporarily unavailable. That effect can be partly undone by using row locks instead of page locks. But that step does not relieve the sweeping effect of a table space lock.

- In a **segmented table space**, rows from different tables are contained in different pages. Locking a page does not lock data from more than one table. Also, DB2 can acquire a table lock, which locks only the data from one specific table. Because a single row, of course, contains data from only one table, the effect of a row lock is the same as for a simple or partitioned table space: it locks one row of data from one table.

- In a **LOB table space**, pages are not locked. Because there is no concept of a row in a LOB table space, rows are not locked. Instead, LOBs are locked. See "LOB locks" on page 353 for more information.

The duration of a lock

Definition
The *duration* of a lock is the length of time the lock is held. It varies according to when the lock is acquired and when it is released.

Effects
For maximum concurrency, locks on a small amount of data held for a short duration are better than locks on a large amount of data held for a long duration. However, acquiring a lock requires processor time, and holding a lock requires storage; thus, acquiring and holding one table space lock is more economical than acquiring and holding many page locks. Consider that trade-off to meet your performance and concurrency objectives.

Duration of partition, table, and table space locks: Partition, table, and table space locks can be acquired when a plan is first allocated, or you can delay acquiring them until the resource they lock is first used. They can be released at the next commit point or be held until the program terminates.

On the other hand, LOB table space locks are always acquired when needed and released at a commit or held until the program terminates. See "LOB locks" on page 353 for information about locking LOBs and LOB table spaces.

Duration of page and row locks: If a page or row is locked, DB2 acquires the lock only when it is needed. When the lock is released depends on many factors, but it is rarely held beyond the next commit point.

For information about controlling the duration of locks, see "Bind options" on page 337.

The mode of a lock

Definition
The *mode* (sometimes *state*) of a lock tells what access to the locked object is permitted to the lock owner and to any concurrent processes.

The possible modes for page and row locks and the modes for partition, table, and table space locks are listed below. See "LOB locks" on page 353 for more information about modes for LOB locks and locks on LOB table spaces.

When a page or row is locked, the table, partition, or table space containing it is also locked. In that case, the table, partition, or table space lock has one of the *intent* modes: IS, IX, or SIX. The modes S, U, and X of table, partition, and table space locks are sometimes called *gross* modes. In the context of reading, SIX is a gross mode lock because you don't get page or row locks; in this sense, it is like an S lock.

Example: An SQL statement locates John Smith in a table of customer data and changes his address. The statement locks the entire table space in mode IX and the specific row that it changes in mode X.

Modes of page and row locks
Modes and their effects are listed in the order of increasing control over resources.

S (SHARE)   The lock owner and any concurrent processes can read, but not change, the locked page or row. Concurrent processes can acquire S or U locks on the page or row or might read data without acquiring a page or row lock.

U (UPDATE)   The lock owner can read, but not change, the locked page or row. Concurrent processes can acquire S locks or might read data without acquiring a page or row lock, but no concurrent process can acquire a U lock.

U locks reduce the chance of deadlocks when the lock owner is reading a page or row to determine whether to change it, because the owner can start with the U lock and then promote the lock to an X lock to change the page or row.

X (EXCLUSIVE)
The lock owner can read or change the locked page or row. A concurrent process can access the data if the process runs with UR isolation. (A concurrent process that is bound with cursor stability and CURRENTDATA(NO) can also read X-locked data if DB2 can tell that the data is committed.)

Modes of table, partition, and table space locks

Modes and their effects are listed in the order of increasing control over resources.

IS (INTENT SHARE)   The lock owner can read data in the table, partition, or table space, but not change it. Concurrent processes can both read and change the data. The lock owner might acquire a page or row lock on any data it reads.

IX (INTENT EXCLUSIVE)   The lock owner and concurrent processes can read and change data in the table, partition, or table space. The lock owner might acquire a page or row lock on any data it reads; it must acquire one on any data it changes.

S (SHARE)   The lock owner and any concurrent processes can read, but not change, data in the table, partition, or table space. The lock owner does not need page or row locks on data it reads.

U (UPDATE)   The lock owner can read, but not change, the locked data; however, the owner can promote the lock to an X lock and then can change the data. Processes concurrent with the U lock can acquire S locks and read the data, but no concurrent process can acquire a U lock. The lock owner does not need page or row locks.

     U locks reduce the chance of deadlocks when the lock owner is reading data to determine whether to change it. U locks are acquired on a table space when locksize is TABLESPACE and the statement is SELECT FOR UPDATE OF. Similarly, U locks are acquired on a table when lock size is TABLE and the statement is SELECT FOR UPDATE OF.

SIX (SHARE with INTENT EXCLUSIVE)
     The lock owner can read and change data in the table, partition, or table space. Concurrent processes can read data in the table, partition, or table space, but not change it. Only when the lock owner changes data does it acquire page or row locks.

X (EXCLUSIVE)   The lock owner can read or change data in the table, partition, or table space. A concurrent process can access the data if the process runs with UR isolation or if data in a LOCKPART(YES) table space is running with CS isolation and CURRENTDATA((NO). The lock owner does not need page or row locks.

Lock mode compatibility

The major effect of the lock mode is to determine whether one lock is compatible with another.

Definition: Locks of some modes do not shut out all other users. Assume that application process A holds a lock on a table space that process B also wants to access. DB2 requests, on behalf of B, a lock of some particular mode. If the mode of A's lock permits B's request, the two locks (or modes) are said to be *compatible*.

Effects of incompatibility: If the two locks are not compatible, B cannot proceed. It must wait until A releases its lock. (And, in fact, it must wait until all existing incompatible locks are released.)

Compatible lock modes: Compatibility for page and row locks is easy to define. Table 39 shows whether page locks of any two modes, or row locks of any two modes, are compatible (Yes) or not (No). No question of compatibility of a page lock with a row lock can arise, because a table space cannot use both page and row locks.

Table 39. Compatibility of page lock and row lock modes

Lock Mode	S	U	X
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

Compatibility for table space locks is slightly more complex. Table 40 shows whether or not table space locks of any two modes are compatible.

Table 40. Compatibility of table and table space (or partition) lock modes

Lock Mode	IS	IX	S	U	SIX	X
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	Yes	No	No
U	Yes	No	Yes	No	No	No
SIX	Yes	No	No	No	No	No
X	No	No	No	No	No	No

The object of a lock

Definition and examples

The *object* of a lock is the resource being locked.

You might have to consider locks on any of the following objects:

- **User data in target tables**. A *target table* is a table that is accessed specifically in an SQL statement, either by name or through a view. Locks on those tables are the most common concern, and the ones over which you have most control.
- **User data in related tables**. Operations subject to referential constraints can require locks on related tables. For example, if you delete from a parent table, DB2 might delete rows from the dependent table as well. In that case, DB2 locks data in the dependent table as well as in the parent table.

  Similarly, operations on rows that contain LOB values might require locks on the LOB table space and possibly on LOB values within that table space. See "LOB locks" on page 353 for more information.

  If your application uses triggers, any triggered SQL statements can cause additional locks to be acquired.
- **DB2 internal objects**. Most of these you are never aware of, but you might notice the following locks on internal objects:
  - Portions of the **DB2 catalog**
  - The **skeleton cursor table** (SKCT) representing an application plan
  - The **skeleton package table** (SKPT) representing a package
  - The **database descriptor** (DBD) representing a DB2 database

For information about any of those, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Indexes and data-only locking

No index page locks are acquired during processing. Instead, DB2 uses a technique called *data-only locking* to serialize changes. Index page latches are acquired to serialize changes within a page and guarantee that the page is physically consistent. Acquiring page latches ensures that transactions accessing the same index page concurrently do not see the page in a partially changed state.

The underlying data page or row locks are acquired to serialize the reading and updating of index entries to ensure the data is logically consistent, meaning that the data is committed and not subject to rollback or abort. The data locks can be held for a long duration such as until commit. However, the page latches are only held for a short duration while the transaction is accessing the page. Because the index pages are not locked, hot spot insert scenarios (which involve several transactions trying to insert different entries into the same index page at the same time) do not cause contention problems in the index.

A query that uses index-only access might lock the data page or row, and that lock can contend with other processes that lock the data. However, using lock avoidance techniques can reduce the contention. See "Lock avoidance" on page 346 for more information about lock avoidance.

Lock tuning

This section describes what you can change to affect how a particular application uses transaction locks, under:
- "Bind options"
- "Isolation overriding with SQL statements" on page 349
- "The statement LOCK TABLE" on page 350

Bind options

These options determine when an application process acquires and releases its locks and to what extent it isolates its actions from possible effects of other processes acting concurrently.

These options of bind operations are relevant to transaction locks:
- "The ACQUIRE and RELEASE options"
- "The ISOLATION option" on page 341
- "The CURRENTDATA option" on page 345

The ACQUIRE and RELEASE options

Effects: The ACQUIRE and RELEASE options of bind determine when DB2 locks an object (table, partition, or table space) your application uses and when it releases the lock. (The ACQUIRE and RELEASE options do not affect page, row, or LOB locks.) The options apply to static SQL statements, which are bound before your program executes. If your program executes dynamic SQL statements, the objects they lock are locked when first accessed and released at the next commit point though some locks acquired for dynamic SQL may be held past commit points. See "The RELEASE option and dynamic statement caching" on page 338.

Option   **Effect**

ACQUIRE(ALLOCATE)	Acquires the lock when the object is allocated. This option is not allowed for BIND or REBIND PACKAGE.
ACQUIRE(USE)	Acquires the lock when the object is first accessed.
RELEASE(DEALLOCATE)	Releases the lock when the object is deallocated (the application ends). The value has no effect on dynamic SQL statements, which always use RELEASE(COMMIT), unless you are using dynamic statement caching. For information about the RELEASE option with dynamic statement caching, see "The RELEASE option and dynamic statement caching".
RELEASE(COMMIT)	Releases the lock at the next commit point, unless there are held cursors or held locators. If the application accesses the object again, it must acquire the lock again.

Example: An application selects employee names and telephone numbers from a table, according to different criteria. Employees can update their own telephone numbers. They can perform several searches in succession. The application is bound with the options ACQUIRE(USE) and RELEASE(DEALLOCATE), for these reasons:

- The alternative to ACQUIRE(USE), ACQUIRE(ALLOCATE), gets a lock of mode IX on the table space as soon as the application starts, because that is needed if an update occurs. But most uses of the application do not update the table and so need only the less restrictive IS lock. ACQUIRE(USE) gets the IS lock when the table is first accessed, and DB2 promotes the lock to mode IX if that is needed later.

- Most uses of this application do not update and do not commit. For those uses, there is little difference between RELEASE(COMMIT) and RELEASE(DEALLOCATE). But administrators might update several phone numbers in one session with the application, and the application commits after each update. In that case, RELEASE(COMMIT) releases a lock that DB2 must acquire again immediately. RELEASE(DEALLOCATE) holds the lock until the application ends, avoiding the processing needed to release and acquire the lock several times.

Effect of LOCKPART YES: Partition locks follow the same rules as table space locks, and *all* partitions are held for the same duration. Thus, if one package is using RELEASE(COMMIT) and another is using RELEASE(DEALLOCATE), all partitions use RELEASE(DEALLOCATE).

The RELEASE option and dynamic statement caching: Generally, the RELEASE option has no effect on dynamic SQL statements with one exception. When you use the bind options RELEASE(DEALLOCATE) and KEEPDYNAMIC(YES), and your subsystem is installed with YES for field CACHE DYNAMIC SQL on panel DSNTIP4, DB2 retains prepared SELECT, INSERT, UPDATE, and DELETE statements in memory past commit points. For this reason, DB2 can honor the RELEASE(DEALLOCATE) option for these dynamic statements. The locks are held until deallocation, or until the commit after the prepared statement is freed from memory, in the following situations:

- The application issues a PREPARE statement with the same statement identifier.
- The statement is removed from memory because it has not been used.

- An object that the statement is dependent on is dropped or altered, or a privilege needed by the statement is revoked.
- RUNSTATS is run against an object that the statement is dependent on.

If a lock is to be held past commit and it is an S, SIX, or X lock on a table space or a table in a segmented table space, DB2 sometimes demotes that lock to an intent lock (IX or IS) at commit. DB2 demotes a gross lock if it was acquired for one of the following reasons:

- DB2 acquired the gross lock because of lock escalation.
- The application issued a LOCK TABLE.
- The application issued a mass delete (DELETE FROM ... without a WHERE clause).

For table spaces defined as LOCKPART YES, lock demotion occurs as with other table spaces; that is, the lock is demoted at the table space level, not the partition level.

Recommendation: Choose a combination of values for ACQUIRE and RELEASE based on the characteristics of the particular application.

Advantages and disadvantages of the combinations

ACQUIRE(ALLOCATE) / RELEASE(DEALLOCATE): In some cases, this combination can avoid deadlocks by locking all needed resources as soon as the program starts to run. This combination is most useful for a long-running application that runs for hours and accesses various tables, because it prevents an untimely deadlock from wasting that processing.

- All tables or table spaces used in DBRMs bound directly to the plan are locked when the plan is allocated.
- All tables or table spaces are unlocked only when the plan terminates.
- The locks used are the most restrictive needed to execute all SQL statements in the plan regardless of whether the statements are actually executed.
- Restrictive states are not checked until the page set is accessed. Locking when the plan is allocated insures that the job is compatible with other SQL jobs. Waiting until the first access to check restrictive states provides greater availability; however, it is possible that an SQL transaction could:
  - Hold a lock on a table space or partition that is stopped
  - Acquire a lock on a table space or partition that is started for DB2 utility access only (ACCESS(UT))
  - Acquire an exclusive lock (IX, X) on a table space or partition that is started for read access only (ACCESS(RO)), thus prohibiting access by readers

Disadvantages: This combination reduces concurrency. It can lock resources in high demand for longer than needed. Also, the option ACQUIRE(ALLOCATE) turns off selective partition locking; if you are accessing a table space defined with LOCKPART YES, all partitions are locked.

Restriction: This combination is not allowed for BIND PACKAGE. Use this combination if processing efficiency is more important than concurrency. It is a good choice for batch jobs that would release table and table space locks only to reacquire them almost immediately. It might even improve concurrency, by allowing batch jobs to finish sooner. Generally, do not use this combination if your application contains many SQL statements that are often not executed.

ACQUIRE(USE) / RELEASE(DEALLOCATE): This combination results in the most efficient use of processing time in most cases.

- A table, partition, or table space used by the plan or package is locked only if it is needed while running.
- All tables or table spaces are unlocked only when the plan terminates.
- The least restrictive lock needed to execute each SQL statement is used, with the exception that if a more restrictive lock remains from a previous statement, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(USE) / RELEASE(COMMIT): This combination is the default combination and provides the greatest concurrency, but it requires more processing time if the application commits frequently.

- A table or table space is locked only when needed. That locking is important if the process contains many SQL statements that are rarely used or statements that are intended to access data only in certain circumstances.
- All tables and table spaces are unlocked when:

TSO, Batch, and CAF

An SQL COMMIT or ROLLBACK statement is issued, or your application process terminates

---

IMS

A CHKP or SYNC call (for single-mode transactions), a GU call to the I/O PCB, or a ROLL or ROLB call is completed

---

CICS

A SYNCPOINT command is issued.

Exception: If the cursor is defined WITH HOLD, table or table space locks necessary to maintain cursor position are held past the commit point. (See "The effect of WITH HOLD for a cursor" on page 348 for more information.

- The least restrictive lock needed to execute each SQL statement is used except when a more restrictive lock remains from a previous statement. In that case, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(ALLOCATE) / RELEASE(COMMIT): This combination is not allowed; it results in an error message from BIND.

The ISOLATION option

Effects: Specifies the degree to which operations are isolated from the possible effects of other operations acting concurrently. Based on this information, DB2 releases S and U locks on rows or pages as soon as possible.

Recommendations: Choose a value of ISOLATION based on the characteristics of the particular application.

Advantages and disadvantages of the isolation values

The various isolation levels offer less or more concurrency at the cost of more or less protection from other application processes. The values you choose should be based primarily on the needs of the application. This section presents the isolation levels in order from the one offering the least concurrency (RR) to that offering the most (UR).

ISOLATION (RR)

Allows the application to read the same pages or rows more than once without allowing any UPDATE, INSERT, or DELETE by another process. All accessed rows or pages are locked, even if they do not satisfy the predicate.

Figure 116 shows that all locks are held until the application commits. In the following example, the rows held by locks L2 and L4 satisfy the predicate.
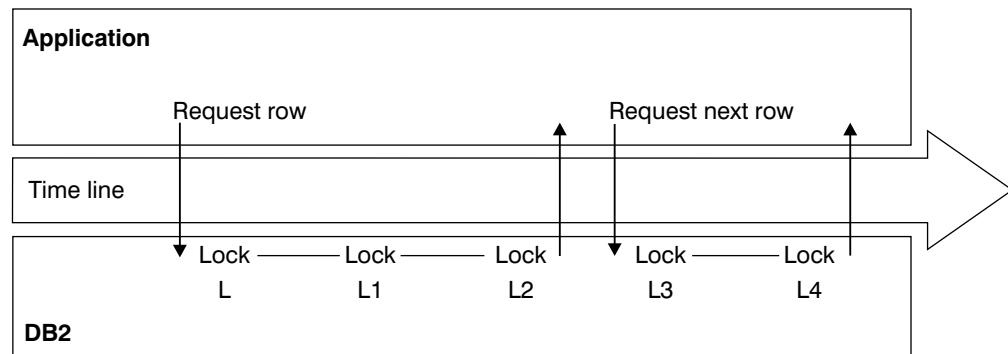
Figure 116. How an application using RR isolation acquires locks. All locks are held until the application commits.

Applications that use repeatable read can leave rows or pages locked for longer periods, especially in a distributed environment, and they can claim more logical partitions than similar applications using cursor stability.

They are also subject to being drained more often by utility operations.

Because so many locks can be taken, lock escalation might take place. Frequent commits release the locks and can help avoid lock escalation.

With repeatable read, lock promotion occurs for table space scan to prevent the insertion of rows that might qualify for the predicate. (If access is via index, DB2 locks the key range. If access is via table space scans, DB2 locks the table, partition, or table space.)

ISOLATION (RS)

Allows the application to read the same pages or rows more than once without allowing qualifying rows to be updated or deleted by another

process. It offers possibly greater concurrency than repeatable read, because although other applications cannot change rows that are returned to the original application, they can insert new rows or update rows that did not satisfy the original application's search condition. Only those rows or pages that satisfy the stage 1 predicate (and all rows or pages evaluated during stage 2 processing) are locked until the application commits. Figure 117 illustrates this. In the example, the rows held by locks L2 and L4 satisfy the predicate.
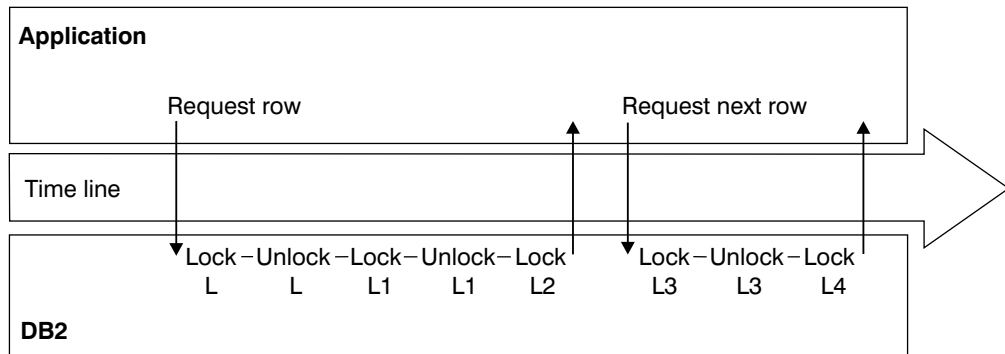
Figure 117. How an application using RS isolation acquires locks when no lock avoidance techniques are used. Locks L2 and L4 are held until the application commits. The other locks aren't held.

Applications using read stability can leave rows or pages locked for long periods, especially in a distributed environment.

If you do use read stability, plan for frequent commit points.

ISOLATION (CS)

Allows maximum concurrency with data integrity. However, after the process leaves a row or page, another process can change the data. With CURRENTDATA(NO), the process doesn't have to leave a row or page to allow another process to change the data. If the first process returns to read the same row or page, the data is not necessarily the same. Consider these consequences of that possibility:

- For table spaces created with LOCKSIZE ROW, PAGE, or ANY, a change can occur even while executing a single SQL statement, if the statement reads the same row more than once. In the following example:

```
SELECT * FROM T1
  WHERE COL1 = (SELECT MAX(COL1) FROM T1);
```

data read by the inner SELECT can be changed by another transaction before it is read by the outer SELECT. Therefore, the information returned by this query might be from a row that is no longer the one with the maximum value for COL1.

- In another case, if your process reads a row and returns later to update it, that row might no longer exist or might not exist in the state that it did when your application process originally read it. That is, another application might have deleted or updated the row. **If your application is doing non-cursor operations on a row under the cursor, make sure the application can tolerate "not found" conditions**.

  Similarly, assume another application updates a row after you read it. If your process returns later to update it based on the value you originally read, you are, in effect, erasing the update made by the other process. **If**

you use isolation (CS) with update, your process might need to lock out concurrent updates. One method is to declare a cursor with the clause FOR UPDATE OF.

┌─ **General-use Programming Interface** ─────────────────────────

For packages and plans that contain updatable scrollable cursors, ISOLATION(CS) lets DB2 use *optimistic concurrency control*. DB2 can use optimistic concurrency control to shorten the amount of time that locks are held in the following situations:

* Between consecutive fetch operations
* Between fetch operations and subsequent positioned update or delete operations

Figure 118 and Figure 119 show processing of positioned update and delete operations without optimistic concurrency control and with optimistic concurrency control.
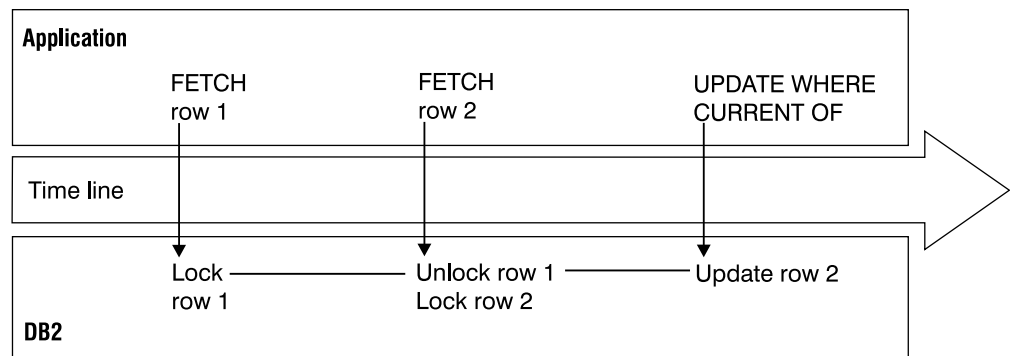
Figure 118. Positioned updates and deletes without optimistic concurrency control
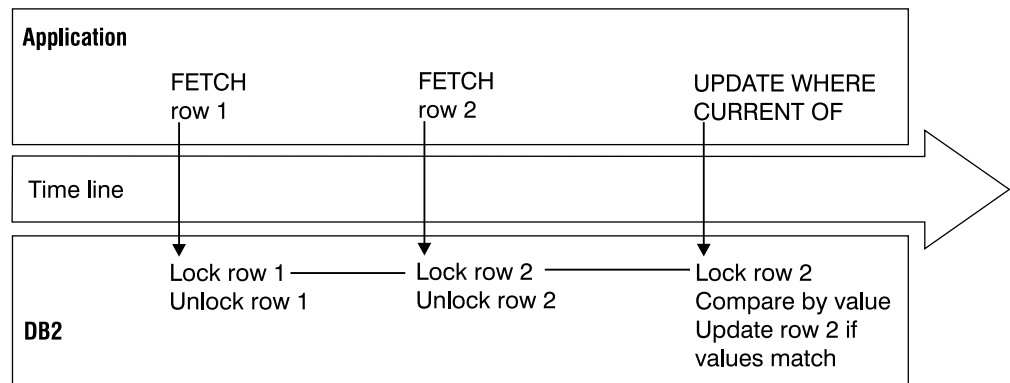
Figure 119. Positioned updates and deletes with optimistic concurrency control

Optimistic concurrency control consists of the following steps:

1. When the application requests a fetch operation to position the cursor on a row, DB2 locks that row, executes the FETCH, and releases the lock.

2. When the application requests a positioned update or delete operation on the row, DB2 performs the following steps:

   a. Locks the row.

b. Reevaluates the predicate to ensure that the row still qualifies for the result table.

   c. For columns that are in the result table, compares current values in the row to the values of the row when step 1 was executed. Performs the positioned update or delete operation only if the values match.

└─ **End of General-use Programming Interface** ────────────

ISOLATION (UR)

Allows the application to read while acquiring few locks, at the risk of reading uncommitted data. UR isolation applies only to read-only operations: SELECT, SELECT INTO, or FETCH from a read-only result table.

There is an element of uncertainty about reading uncommitted data.

Example: An application tracks the movement of work from station to station along an assembly line. As items move from one station to another, the application subtracts from the count of items at the first station and adds to the count of items at the second. Assume you want to query the count of items at all the stations, while the application is running concurrently.

What can happen if your query reads data that the application has changed but has not committed?

   If the application subtracts an amount from one record before adding it to another, *the query could miss the amount entirely.*

   If the application adds first and then subtracts, *the query could add the amount twice*.

If those situations can occur and are unacceptable, do not use UR isolation.

Restrictions: You cannot use UR isolation for the types of statement listed below. If you bind with ISOLATION(UR), and the statement does not specify WITH RR or WITH RS, then DB2 uses CS isolation for:
- INSERT, UPDATE, and DELETE
- Any cursor defined with FOR UPDATE OF

When can you use uncommitted read (UR)? You can probably use UR isolation in cases like the following ones:
- **When errors cannot occur**.

   Example: A reference table, like a table of descriptions of parts by part number. It is rarely updated, and reading an uncommitted update is probably no more damaging than reading the table 5 seconds earlier. Go ahead and read it with ISOLATION(UR).

   Example: The employee table of Spiffy Computer, our hypothetical user. For security reasons, updates can be made to the table only by members of a single department. And that department is also the only one that can query the entire table. It is easy to restrict queries to times when no updates are being made and then run with UR isolation.

- **When an error is acceptable**.

   Example: Spiffy wants to do some statistical analysis on employee data. A typical question is, "What is the average salary by sex within education

level?" Because reading an occasional uncommitted record cannot affect the averages much, UR isolation can be used.

- **When the data already contains inconsistent information**.

  Example: Spiffy gets sales leads from various sources. The data is often inconsistent or wrong, and end users of the data are accustomed to dealing with that. Inconsistent access to a table of data on sales leads does not add to the problem.

  Do NOT use uncommitted read (UR):
  When the computations must balance
  When the answer must be accurate
  When you are not sure it can do no damage

Restrictions on concurrent access: An application using UR isolation cannot run concurrently with a utility that drains all claim classes. Also, the application must acquire the following locks:

- A special *mass delete lock* acquired in S mode on the target table or table space. A "mass delete" is a DELETE statement without a WHERE clause; that operation must acquire the lock in X mode and thus cannot run concurrently.

- An IX lock on any table space used in the work file database. That lock prevents dropping the table space while the application is running.

- If LOB values are read, LOB locks and a lock on the LOB table space. If the LOB lock is not available because it is held by another application in an incompatible lock state, the UR reader skips the LOB and moves on to the next LOB that satisfies the query.

The CURRENTDATA option

The CURRENTDATA option has different effects, depending on if access is local or remote:

- For **local** access, the option tells whether the data upon which your cursor is positioned must remain identical to (or "current with") the data in the local base table. For cursors positioned on data in a work file, the CURRENTDATA option has no effect. This effect only applies to read-only or *ambiguous* cursors in plans or packages bound with CS isolation.

  A cursor is "ambiguous" if DB2 cannot tell whether it is used for update or read-only purposes. If the cursor appears to be used only for read-only, but dynamic SQL could modify data through the cursor, then the cursor is ambiguous. If you use CURRENTDATA to indicate an ambiguous cursor is read-only when it is actually targeted by dynamic SQL for modification, you'll get an error. See "Problems with ambiguous cursors" on page 347 for more information about ambiguous cursors.

- For a request to a **remote** system, CURRENTDATA has an effect for ambiguous cursors using isolation levels RR, RS, or CS. For ambiguous cursors, it turns block fetching on or off. (Read-only cursors and UR isolation always use block fetch.) Turning on block fetch offers best performance, but it means the cursor is not current with the base table at the remote site.

Local access: Locally, **CURRENTDATA(YES)** means that the data upon which the cursor is positioned cannot change while the cursor is positioned on it. If the cursor is positioned on data in a local base table or index, then the data returned with the cursor is current with the contents of that table or index. If the cursor is positioned on data in a work file, the data returned with the cursor is current only with the contents of the work file; it is not necessarily current with the contents of the underlying table or index.

Figure 120 shows locking with CURRENTDATA(YES).
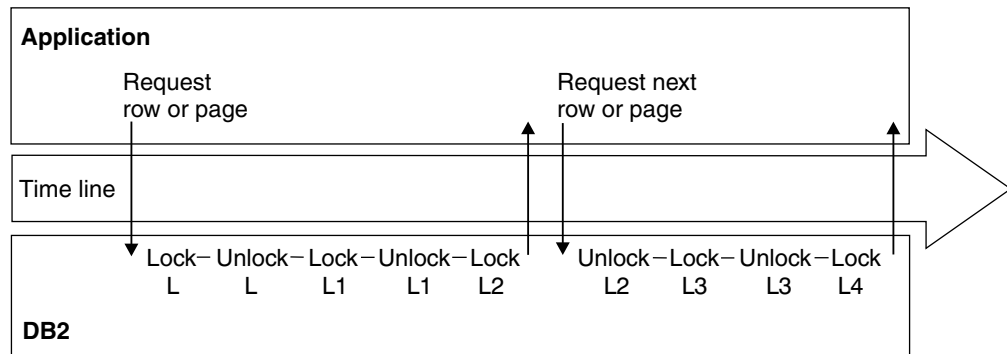
Figure 120. How an application using isolation CS with CURRENTDATA(YES) acquires locks. This figure shows access to the base table. The L2 and L4 locks are released after DB2 moves to the next row or page. When the application commits, the last lock is released.

As with work files, if a cursor uses query parallelism, data is not necessarily current with the contents of the table or index, regardless of whether a work file is used. Therefore, for work file access or for parallelism on read-only queries, the CURRENTDATA option has no effect.

If you are using parallelism but want to maintain currency with the data, you have the following options:

- Disable parallelism (Use SET DEGREE = '1' or bind with DEGREE(1)).
- Use isolation RR or RS (parallelism can still be used).
- Use the LOCK TABLE statement (parallelism can still be used).

For local access, **CURRENTDATA(NO)** is similar to CURRENTDATA(YES) except for the case where a cursor is accessing a base table rather than a result table in a work file. In those cases, although CURRENTDATA(YES) can guarantee that the cursor and the base table are current, CURRENTDATA(NO) makes no such guarantee.

Remote access: For access to a remote table or index, **CURRENTDATA(YES)** turns off block fetching for ambiguous cursors. The data returned with the cursor is current with the contents of the remote table or index for ambiguous cursors. See "Use block fetch" on page 383 for more information about the effect of CURRENTDATA on block fetch.

Lock avoidance: With CURRENTDATA(NO), you have much greater opportunity for avoiding locks. DB2 can test to see if a row or page has committed data on it. If it has, DB2 does not have to obtain a lock on the data at all. Unlocked data is returned to the application, and the data can be changed while the cursor is positioned on the row. (For SELECT statements in which no cursor is used, such as those that return a single row, a lock is not held on the row unless you specify WITH RS or WITH RR on the statement.)

To take the best advantage of this method of avoiding locks, make sure all applications that are accessing data concurrently issue COMMITs frequently.

Figure 121 on page 347 shows how DB2 can avoid taking locks and Table 41 on page 347 summarizes the factors that influence lock avoidance.
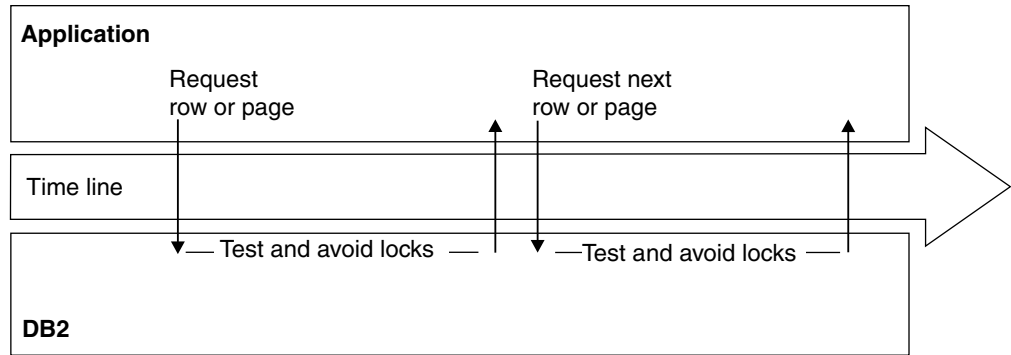
Figure 121. Best case of avoiding locks using CS isolation with CURRENTDATA(NO). This figure shows access to the base table. If DB2 must take a lock, then locks are released when DB2 moves to the next row or page, or when the application commits (the same as CURRENTDATA(YES)).

Table 41. Lock avoidance factors. "Returned data" means data that satisfies the predicate. "Rejected data" is that which does not satisfy the predicate.

Isolation	CURRENTDATA	Cursor type	Avoid locks on returned data?	Avoid locks on rejected data?
UR	N/A	Read-only	N/A	N/A
CS	YES	Read-only	No	Yes
		Updatable		
		Ambiguous		
	NO	Read-only	Yes	
		Updatable	No	
		Ambiguous	Yes	
RS	N/A	Read-only	No	Yes
		Updatable		
		Ambiguous		
RR	N/A	Read-only	No	No
		Updatable		
		Ambiguous		

Problems with ambiguous cursors: As shown in Table 41, ambiguous cursors can sometimes prevent DB2 from using lock avoidance techniques. However, misuse of an ambiguous cursor can cause your program to receive a -510 SQLCODE:

- The plan or package is bound with CURRENTDATA(NO)
- An OPEN CURSOR statement is performed *before* a dynamic DELETE WHERE CURRENT OF statement against that cursor is prepared
- One of the following conditions is true for the open cursor:
  - Lock avoidance is successfully used on that statement.
  - Query parallelism is used.
  - The cursor is distributed, and block fetching is used.

In all cases, it is a good programming technique to eliminate the ambiguity by declaring the cursor with one of the clauses FOR FETCH ONLY or FOR UPDATE OF.

When plan and package options differ

A plan bound with one set of options can include packages in its package list that were bound with different sets of options. In general, statements in a DBRM bound as a package use the options that the package was bound with, and statements in DBRMs bound to a plan use the options that the plan was bound with.

For example, the plan value for CURRENTDATA has no effect on the packages executing under that plan. If you do not specify a CURRENTDATA option explicitly when you bind a package, the default is CURRENTDATA(YES).

The rules are slightly different for the bind options RELEASE and ISOLATION. The values of those two options are set when the lock on the resource is acquired and usually stay in effect until the lock is released. But a conflict can occur if a statement that is bound with one pair of values requests a lock on a resource that is already locked by a statement that is bound with a different pair of values. DB2 resolves the conflict by resetting each option with the available value that causes the lock to be held for the greatest duration.

If the conflict is between RELEASE(COMMIT) and RELEASE(DEALLOCATE), then the value used is RELEASE(DEALLOCATE).

Table 42 shows how conflicts between isolation levels are resolved. The first column is the existing isolation level, and the remaining columns show what happens when another isolation level is requested by a new application process.

Table 42. Resolving isolation conflicts

	UR	CS	RS	RR
UR	n/a	CS	RS	RR
CS	CS	n/a	RS	RR
RS	RS	RS	n/a	RR
RR	RR	RR	RR	n/a

The effect of WITH HOLD for a cursor

For a cursor defined as WITH HOLD, the cursor position is maintained past a commit point. Hence, locks and claims needed to maintain that position are not released immediately, even if they were acquired with ISOLATION(CS) or RELEASE(COMMIT).

For locks and claims needed for cursor position, the rules described above differ as follows:

Page and row locks: If your installation specifies NO on the RELEASE LOCKS field of installation panel DSNTIP4, as described in Part 5 (Volume 2) of *DB2 Administration Guide*, a page or row lock is held past the commit point. This page or row lock is not necessary for cursor position, but the NO option is provided for compatibility that might rely on this lock. However, an X or U lock is demoted to an S lock at that time. (Because changes have been committed, exclusive control is no longer needed.) After the commit point, the lock is released at the next commit point, provided that no cursor is still positioned on that page or row.

A YES for RELEASE LOCKS means that no data page or row locks are held past commit.

Table, table space, and DBD locks: All necessary locks are held past the commit point. After that, they are released according to the RELEASE option under which they were acquired: for COMMIT, at the next commit point after the cursor is closed; for DEALLOCATE, when the application is deallocated.

Claims: All claims, for any claim class, are held past the commit point. They are released at the next commit point after all held cursors have moved off the object or have been closed.

Isolation overriding with SQL statements

Function of the WITH clause: You can override the isolation level with which a plan or package is bound by the WITH clause on certain SQL statements.

Example: This statement:
```
SELECT MAX(BONUS), MIN(BONUS), AVG(BONUS)
  INTO :MAX, :MIN, :AVG
  FROM DSN8710.EMP
     WITH UR;
```

finds the maximum, minimum, and average bonus in the sample employee table. The statement is executed with uncommitted read isolation, regardless of the value of ISOLATION with which the plan or package containing the statement is bound.

Rules for the WITH clause: The WITH clause:
- Can be used on these statements:
  - Select-statement
  - SELECT INTO
  - Searched delete
  - INSERT from fullselect
  - Searched update
- Cannot be used on subqueries.
- Can specify the isolation levels that specifically apply to its statement. (For example, because WITH UR applies only to read-only operations, you cannot use it on an INSERT statement.)
- Overrides the isolation level for the plan or package only for the statement in which it appears.

Using KEEP UPDATE LOCKS on the WITH clause: You can use the clause KEEP UPDATE LOCKS clause when you specify a SELECT with FOR UPDATE OF. This option is only valid when you use WITH RR or WITH RS. By using this clause, you tell DB2 to acquire an X lock instead of an U or S lock on all the qualified pages or rows.

Here is an example:
```
SELECT ...
 FOR UPDATE OF WITH RS KEEP UPDATE LOCKS;
```

With read stability (RS) isolation, a row or page rejected during stage 2 processing still has the X lock held on it, even though it is not returned to the application.

With repeatable read (RR) isolation, DB2 acquires the X locks on all pages or rows that fall within the range of the selection expression.

All X locks are held until the application commits. Although this option can reduce concurrency, it can prevent some types of deadlocks and can better serialize access to data.

The statement LOCK TABLE

For information about using LOCK TABLE on an auxiliary table, see "The LOCK TABLE statement" on page 356.

The purpose of LOCK TABLE

Use the LOCK TABLE statement to override DB2's rules for choosing initial lock attributes. Two examples are:

```
LOCK TABLE table-name IN SHARE MODE;
LOCK TABLE table-name PART n IN EXCLUSIVE MODE;
```

Executing the statement requests a lock immediately, unless a suitable lock exists already, as described below. The bind option RELEASE determines when locks acquired by LOCK TABLE or LOCK TABLE with the PART option are released.

You can use LOCK TABLE on any table, including auxiliary tables of LOB table spaces. See "The LOCK TABLE statement" on page 356 for information about locking auxiliary tables.

LOCK TABLE has no effect on locks acquired at a remote server.

The effect of LOCK TABLE

Table 43 shows the modes of locks acquired in segmented and nonsegmented table spaces for the SHARE and EXCLUSIVE modes of LOCK TABLE. Auxiliary tables of LOB table spaces are considered nonsegmented table spaces and have the same locking behavior.

Table 43. Modes of locks acquired by LOCK TABLE. LOCK TABLE on partitions behave the same as nonsegmented table spaces.

LOCK TABLE IN	Nonsegmented Table Space	Segmented Table Space	
		Table	Table Space
EXCLUSIVE MODE	X	X	IX
SHARE MODE	S or SIX	S or SIX	IS

Note: The SIX lock is acquired if the process already holds an IX lock. SHARE MODE has no effect if the process already has a lock of mode SIX, U, or X.

Recommendations for using LOCK TABLE

Use LOCK TABLE to prevent other application processes from changing any row in a table or partition that your process is accessing. For example, suppose that you access several tables. You can tolerate concurrent updates on all the tables except one; for that one, you need RR or RS isolation. There are several ways to handle the situation:

- Bind the application plan with RR or RS isolation. But that affects all the tables you access and might reduce concurrency.
- Design the application to use packages and access the exceptional table in only a few packages. Bind those packages with RR or RS isolation and the plan with CS isolation. Only the tables accessed within those packages are accessed with RR or RS isolation.

- Add the clause WITH RR or WITH RS to statements that must be executed with RR or RS isolation. Statements that do not use WITH are executed as specified by the bind option ISOLATION.
- Bind the application plan with CS isolation *and* execute LOCK TABLE for the exceptional table. (If there are other tables in the same table space, see the caution that follows.) LOCK TABLE locks out changes by any other process, giving the exceptional table a degree of isolation even more thorough than repeatable read. All tables in other table spaces are shared for concurrent update.

Caution when using LOCK TABLE with simple table spaces: The statement locks all tables in a simple table space, even though you name only one table. No other process can update the table space for the duration of the lock. If the lock is in exclusive mode, no other process can read the table space, unless that process is running with UR isolation.

Additional examples of LOCK TABLE: You might want to lock a table or partition that is normally shared for any of the following reasons:

Taking a"snapshot"
> If you want to access an entire table throughout a unit of work as it was at a particular moment, you must lock out concurrent changes. If other processes can access the table, use LOCK TABLE IN SHARE MODE. (RR isolation is not enough; it locks out changes only from rows or pages you have already accessed.)

Avoiding overhead
> If you want to update a large part of a table, it can be more efficient to prevent concurrent access than to lock each page as it is updated and unlock it when it is committed. Use LOCK TABLE IN EXCLUSIVE MODE.

Preventing timeouts
> Your application has a high priority and must not risk timeouts from contention with other application processes. Depending on whether your application updates or not, use either LOCK IN EXCLUSIVE MODE or LOCK TABLE IN SHARE MODE.

Access paths

The access path used can affect the mode, size, and even the object of a lock. For example, an UPDATE statement using a table space scan might need an X lock on the entire table space. If rows to be updated are located through an index, the same statement might need only an IX lock on the table space and X locks on individual pages or rows.

If you use the EXPLAIN statement to investigate the access path chosen for an SQL statement, then check the lock mode in column TSLOCKMODE of the resulting PLAN_TABLE. If the table resides in a nonsegmented table space, or is defined with LOCKSIZE TABLESPACE, the mode shown is that of the table space lock. Otherwise, the mode is that of the table lock.

The important points about DB2 locks:

- You usually do not have to lock data explicitly in your program.
- DB2 ensures that your program does not retrieve uncommitted data unless you specifically allow that.
- Any page or row where your program updates, inserts, or deletes stays locked at least until the end of a unit of work, regardless of the isolation level. No other process can access the object in any way until then, unless you specifically allow that access to that process.
- Commit often for concurrency. Determine points in your program where changed data is consistent. At those points, issue:

> **TSO, Batch, and CAF**
> An SQL COMMIT statement

> **IMS**
> A CHKP or SYNC call, or (for single-mode transactions) a GU call to the I/O PCB

> **CICS**
> A SYNCPOINT command.

- Bind with ACQUIRE(USE) to improve concurrency.
- Set ISOLATION (usually RR, RS, or CS) when you bind the plan or package.
    - With RR (repeatable read), all accessed pages or rows are locked until the next commit point. (See "The effect of WITH HOLD for a cursor" on page 348 for information about cursor position locks for cursors defined WITH HOLD.)
    - With RS (read stability), all qualifying pages or rows are locked until the next commit point. (See "The effect of WITH HOLD for a cursor" on page 348 for information about cursor position locks for cursors defined WITH HOLD.)
    - With CS (cursor stability), only the pages or rows currently accessed can be locked, and those locks might be avoided. (You can access one page or row for each open cursor.)
- You can also set isolation for specific SQL statements, using WITH.
- A deadlock can occur if two processes each hold a resource that the other needs. One process is chosen as "victim", its unit of work is rolled back, and an SQL error code is issued.

Figure 122. Summary of DB2 locks (Part 1 of 2)

- You can lock an entire nonsegmented table space, or an entire table in a segmented table space, by the statement LOCK TABLE:
  - To let other users retrieve, but not update, delete, or insert, issue:

    ```
    LOCK TABLE table-name IN SHARE MODE
    ```
  - To prevent other users from accessing rows in any way, except by using UR isolation, issue:

    ```
    LOCK TABLE table-name IN EXCLUSIVE MODE
    ```

Figure 122. Summary of DB2 locks (Part 2 of 2)

LOB locks

The locking activity for LOBs is described separately from transaction locks because the purpose of LOB locks is different than that of regular transaction locks.

Terminology: A lock that is taken on a LOB value in a LOB table space is called a **LOB lock**.

In this section: The following topics are described:
- "Relationship between transaction locks and LOB locks"
- "Hierarchy of LOB locks" on page 354
- "LOB and LOB table space lock modes" on page 355
- "Duration of locks" on page 355
- "Instances when locks on LOB table space are not taken" on page 356
- "The LOCK TABLE statement" on page 356

Relationship between transaction locks and LOB locks

As described in "Introduction to LOBs" on page 229, LOB column values are stored in a different table space, a LOB table space, from the values in the base table. An application that reads or updates a row in a table that contains LOB columns obtains its normal transaction locks on the base table. The locks on the base table also control concurrency for the LOB table space. When locks are not acquired on the base table, such as for ISO(UR), DB2 maintains data consistency by using locks on the LOB table space. Even when locks are acquired on the base table, DB2 still obtains locks on the LOB table space.

DB2 also obtains locks on the LOB table space and the LOB values stored in that LOB table space, but those locks have the following primary purposes:
- To determine whether space from a deleted LOB can be reused by an inserted or updated LOB

  Storage for a deleted LOB is not reused until no more readers (including held locators) are on the LOB and the delete operation has been committed.
- To prevent deallocating space for a LOB that is currently being read

  A LOB can be deleted from one application's point-of-view while a reader from another application is reading the LOB. The reader continues reading the LOB because all readers, including those readers that are using uncommitted read isolation, acquire S-locks on LOBs to prevent the storage for the LOB they are reading from being deallocated. That lock is held until commit. A held LOB locator or a held cursor cause the LOB lock and LOB table space lock to be held past commit.

In summary, the main purpose of LOB locks is for managing the space used by LOBs and to ensure that LOB readers do not read partially updated LOBs. Applications need to free held locators so that the space can be reused.

Table 44 shows the relationship between the action that is occurring on the LOB value and the associated LOB table space and LOB locks that are acquired.

Table 44. Locks that are acquired for operations on LOBs. This table does not account for gross locks that can be taken because of LOCKSIZE TABLESPACE, the LOCK TABLE statement, or lock escalation.

Action on LOB value	LOB table space lock	LOB lock	Comment
Read (including UR)	IS	S	Prevents storage from being reused while the LOB is being read or while locators are referencing the LOB
Insert	IX	X	Prevents other processes from seeing a partial LOB
Delete	IS	S	To hold space in case the delete is rolled back. (The X is on the base table row or page.) Storage is not reusable until the delete is committed and no other readers of the LOB exist.
Update	IS->IX	Two LOB locks: an S-lock for the delete and an X-lock for the insert.	Operation is a delete followed by an insert.
Update the LOB to null or zero-length	IS	S	No insert, just a delete.
Update a null or zero-length LOB to a value	IX	X	No delete, just an insert.

ISOLATION(UR) or ISOLATION(CS): When an application is reading rows using uncommitted read or lock avoidance, no page or row locks are taken on the base table. Therefore, these readers must take an S LOB lock to ensure that they are not reading a partial LOB or a LOB value that is inconsistent with the base row.

Hierarchy of LOB locks

Just as page locks (or row locks) and table space locks have a a hierarchical relationship, LOB locks and locks on LOB table spaces have a hierarchical relationship. (See Figure 115 on page 332 for a picture of the hierarchical relationship.) If the LOB table space is locked with a gross lock, then LOB locks are not acquired. In a data sharing environment, the lock on the LOB table space is used to determine whether the lock on the LOB must be propagated beyond the local IRLM.

LOB and LOB table space lock modes

Modes of LOB locks
The following LOB lock modes are possible:

S (SHARE)   The lock owner and any concurrent processes can read, update, or delete the locked LOB. Concurrent processes can acquire an S lock on the LOB. The purpose of the S lock is to reserve the space used by the LOB.

X (EXCLUSIVE)
The lock owner can read or change the locked LOB. Concurrent processes cannot access the LOB.

Modes of LOB table space locks
The following locks modes are possible on the LOB table space:

IS (INTENT SHARE)
The lock owner can update LOBs to null or zero-length, or read or delete LOBs in the LOB table space. Concurrent processes can both read and change LOBs in the same table space. The lock owner acquires a LOB lock on any data that it reads or deletes.

IX (INTENT EXCLUSIVE)
The lock owner and concurrent processes can read and change data in the LOB table space. The lock owner acquires a LOB lock on any data it accesses.

S (SHARE)   The lock owner and any concurrent processes can read and delete LOBs in the LOB table space. The lock owner does not need LOB locks.

SIX (SHARE with INTENT EXCLUSIVE)
The lock owner can read and change data in the LOB table space. If the lock owner is inserting (INSERT or UPDATE), the lock owner obtains a LOB lock. Concurrent processes can read or delete data in the LOB table space (or update to a null or zero-length LOB).

X (EXCLUSIVE)
The lock owner can read or change LOBs in the LOB table space. The lock owner does not need LOB locks. Concurrent processes cannot access the data.

Duration of locks

Duration of locks on LOB table spaces
Locks on LOB table spaces are acquired when they are needed; that is, the ACQUIRE option of BIND has no effect on when the table space lock on the LOB table space is taken. The table space lock is released according to the value specified on the RELEASE option of BIND (except when a cursor is defined WITH HOLD or if a held LOB locator exists).

Duration of LOB locks
Locks on LOBs are taken when they are needed and are usually released at commit. However, if that LOB value is assigned to a LOB locator, the S lock remains until the application commits.

If the application uses HOLD LOCATOR, the LOB lock is not freed until the first commit operation after a FREE LOCATOR statement is issued, or until the thread is deallocated.

A note about held cursors: If a cursor is defined WITH HOLD, LOB locks are held through commit operations.

A note about INSERT with fullselect: Because LOB locks are held until commit and because locks are put on each LOB column in both a source table and a target table, it is possible that a statement such as an INSERT with a fullselect that involves LOB columns can accumulate many more locks than a similar statement that does not involve LOB columns. To prevent system problems caused by too many locks, you can:

- Ensure that you have lock escalation enabled for the LOB table spaces that are involved in the INSERT. In other words, make sure that LOCKMAX is non-zero for those LOB table spaces.
- Alter the LOB table space to change the LOCKSIZE to TABLESPACE before executing the INSERT with fullselect.
- Increase the LOCKMAX value on the table spaces involved and ensure that the user lock limit is sufficient.
- Use LOCK TABLE statements to lock the LOB table spaces. (Locking the auxiliary table that is contained in the LOB table space locks the LOB table space.)

Instances when locks on LOB table space are not taken

A lock might not be acquired on a LOB table space at all. For example, if a row is deleted from a table and the value of the LOB column is null, the LOB table space associated with that LOB column is not locked. DB2 does not access the LOB table space if the application:

- Selects a LOB that is null or zero length
- Deletes a row where the LOB is null or zero length
- Inserts a null or zero length LOB
- Updates a null or zero-length LOB to null or zero-length

The LOCK TABLE statement

"The statement LOCK TABLE" on page 350 describes how and why you might use a LOCK TABLE statement on a table. The reasons for using LOCK TABLE on an auxiliary table are somewhat different than that for regular tables.

- You can use LOCK TABLE to control the number of locks acquired on the auxiliary table.
- You can use LOCK TABLE IN SHARE MODE to prevent other applications from inserting LOBs.

  With auxiliary tables, LOCK TABLE IN SHARE MODE does not prevent any changes to the auxiliary table. The statement does prevent LOBs from being inserted into the auxiliary table, but it does not prevent deletes. Updates are generally restricted also, except where the LOB is updated to a null value or a zero-length string.

- You can use LOCK TABLE IN EXCLUSIVE MODE to prevent other applications from accessing LOBs.

  With auxiliary tables, LOCK TABLE IN EXCLUSIVE MODE also prevents access from uncommitted readers.

- Either statement eliminates the need for lower-level LOB locks.

Chapter 18. Planning for recovery

During recovery, when a DB2 database is restoring to its most recent consistent state, you must back out any uncommitted changes to data that occurred before the program abend or system failure. You must do this without interfering with other system activities.

If your application intercepts abends, DB2 commits work because it is unaware that an abend has occurred. If you want DB2 to roll back work automatically when an abend occurs in your program, do not let the program or runtime environment intercept the abend. For example, if your program uses Language Environment, and you want DB2 to roll back work automatically when an abend occurs in the program, specify the runtime options ABTERMENC(ABEND) and TRAP(ON).

A *unit of work* is a logically distinct procedure containing steps that change the data. If all the steps complete successfully, you want the data changes to become permanent. But, if any of the steps fail, you want all modified data to return to the original value before the procedure began.

For example, suppose two employees in the sample table DSN8710.EMP exchange offices. You need to exchange their office phone numbers in the PHONENO column. You would use two UPDATE statements to make each phone number current. Both statements, taken together, are a unit of work. You want both statements to complete successfully. For example, if only one statement is successful, you want both phone numbers rolled back to their original values before attempting another update.

When a unit of work completes, all locks implicitly acquired by that unit of work after it begins are released, allowing a new unit of work to begin.

The amount of processing time used by a unit of work in your program determines the length of time DB2 prevents other users from accessing that locked data. When several programs try to use the same data concurrently, each program's unit of work must be as short as possible to minimize the interference between the programs. The remainder of this chapter describes the way a unit of work functions in various environments. For more information on unit of work, see Chapter 1 of *DB2 SQL Reference* or Part 4 (Volume 1) of *DB2 Administration Guide*.

Unit of work in TSO (batch and online)

A unit of work starts when the first DB2 object updates occur.

A unit of work ends when one of the following conditions occur:
- The program issues a subsequent COMMIT statement. At this point in the processing, your program is confident the data is consistent; all data changes since the previous commit point were made correctly.
- The program issues a subsequent ROLLBACK statement. At this point in the processing, your program has determined that the data changes were not made correctly and, therefore, does not want to make the data changes permanent.
- The program terminates and returns to the DSN command processor, which returns to the TSO Terminal Monitor Program (TMP).

A *commit point* occurs when you issue a COMMIT statement or your program terminates normally. You should issue a COMMIT statement only when you are sure

the data is in a consistent state. For example, a bank transaction might transfer funds from account A to account B. The transaction first subtracts the amount of the transfer from account A, and then adds the amount to account B. Both events, taken together, are a unit of work. When both events complete (and not before), the data in the two accounts is consistent. The program can then issue a COMMIT statement. A ROLLBACK statement causes any data changes, made since the last commit point, to be backed out.

Before you can connect to another DBMS you must issue a COMMIT statement. If the system fails at this point, DB2 cannot know that your transaction is complete. In this case, as in the case of a failure during a one-phase commit operation for a single subsystem, you must make your own provision for maintaining data integrity.

If your program abends or the system fails, DB2 backs out uncommitted data changes. Changed data returns to its original condition without interfering with other system activities.

Unit of work in CICS

In CICS, all the processing that occurs in your program between two commit points is known as a logical unit of work (LUW) or *unit of work*. Generally, a unit of work is a *sequence* of actions that must complete before any of the *individual* actions in the sequence can complete. For example, the actions of decrementing an inventory file and incrementing a reorder file by the same quantity can constitute a unit of work: *both* steps must complete before either step is complete. (If one action occurs and not the other, the database loses its integrity, or consistency.)

A unit of work is marked as complete by a *commit* or *synchronization (sync)* point, defined as follows:

- Implicitly at the end of a transaction, signalled by a CICS RETURN command at the highest logical level.
- Explicitly by CICS SYNCPOINT commands that the program issues at logically appropriate points in the transaction.
- Implicitly through a DL/I PSB termination (TERM) call or command.
- Implicitly when a batch DL/I program issues a DL/I checkpoint call. This can occur when the batch DL/I program is sharing a database with CICS applications through the database sharing facility.

Consider the inventory example, in which the quantity of items sold is subtracted from the inventory file and then added to the reorder file. When both transactions complete (and not before) and the data in the two files is consistent, the program can then issue a DL/I TERM call or a SYNCPOINT command. If one of the steps fails, you want the data to return to the value it had before the unit of work began. That is, you want it rolled back to a previous point of consistency. You can achieve this by using the SYNCPOINT command with the ROLLBACK option.

By using a SYNCPOINT command with the ROLLBACK option, you can back out uncommitted data changes. For example, a program that updates a set of related rows sometimes encounters an error after updating several of them. The program can use the SYNCPOINT command with the ROLLBACK option to *undo* all of the updates without giving up control.

The SQL COMMIT and ROLLBACK statements are not valid in a CICS environment. You can coordinate DB2 with CICS functions used in programs, so that DB2 and non-DB2 data are consistent.

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original condition without interfering with other system activities. Sometimes, DB2 data does not return to a consistent state immediately. DB2 does not process *indoubt data* (data that is neither uncommitted nor committed) until the CICS attachment facility is also restarted. To ensure that DB2 and CICS are synchronized, restart both DB2 and the CICS attachment facility.

Unit of work in IMS (online)

In IMS, a *unit of work* starts:
* When the program starts
* After a CHKP, SYNC, ROLL, or ROLB call has completed
* For single-mode transactions, when a GU call is issued to the I/O PCB.

A unit of work ends when:
* The program issues a subsequent CHKP or SYNC call, or (for single-mode transactions) issues a GU call to the I/O PCB. At this point in the processing, the data is consistent. All data changes made since the previous commit point are made correctly.
* The program issues a subsequent ROLB or ROLL call. At this point in the processing, your program has determined that the data changes are not correct and, therefore, that the data changes should not become permanent.
* The program terminates.

A *commit point* can occur in a program as the result of any one of the following four events:
* The program terminates normally. Normal program termination is always a commit point.
* The program issues a *checkpoint call*. Checkpoint calls are a program's means of explicitly indicating to IMS that it has reached a commit point in its processing.
* The program issues a *SYNC call*. The SYNC call is a Fast Path system service call to request commit point processing. You can use a SYNC call only in a nonmessage-driven Fast Path program.
* For a program that processes messages as its input, a commit point can occur when the program retrieves a new message. IMS considers a new message the start of a new unit of work in the program. Unless you define the transaction as single- or multiple-mode on the TRANSACT statement of the APPLCTN macro for the program, retrieving a new message does not signal a commit point. For more information about the APPLCTN macro, see the *IMS Install Volume 2: System Definition and Tailoring*.
  * If you specify *single-mode*, a commit point in DB2 occurs each time the program issues a call to retrieve a new message. Specifying single-mode can simplify recovery; you can restart the program from the most recent call for a new message if the program abends. When IMS restarts the program, the program starts by processing the next message.
  * If you specify *multiple-mode*, a commit point occurs when the program issues a checkpoint call or when it terminates normally. Those are the only times during the program that IMS sends the program's output messages to their destinations. Because there are fewer commit points to process in multiple-mode programs than in single-mode programs, multiple-mode programs could perform slightly better than single-mode programs. When a multiple-mode program abends, IMS can restart it only from a checkpoint call. Instead of having only the most recent message to reprocess, a program

might have several messages to reprocess. The number of messages to process depends on when the program issued the last checkpoint call.

DB2 does some processing with single- and multiple-mode programs that IMS does not. When a multiple-mode program issues a call to retrieve a new message, DB2 performs an authorization check and closes all open cursors in the program.

At the time of a commit point:
- IMS and DB2 can release locks that the program has held on data since the last commit point. That makes the data available to other application programs and users. (However, when you define a cursor as WITH HOLD in a BMP program, DB2 holds those locks until the cursor closes or the program ends.)
- DB2 closes any open cursors that the program has been using. Your program must issue CLOSE CURSOR statements *before* a checkpoint call or a GU to the message queue, *not after*.
- IMS and DB2 make the program's changes to the data base permanent.

If the program abends before reaching the commit point:
- Both IMS and DB2 back out all the changes the program has made to the database since the last commit point.
- IMS deletes any output messages that the program has produced since the last commit point (for nonexpress PCBs).

If the program processes messages, IMS sends the output messages that the application program produces to their final destinations. Until the program reaches a commit point, IMS holds the program's output messages at a temporary destination. If the program abends, people at terminals, and other application programs do not receive inaccurate information from the terminating application program.

The SQL COMMIT and ROLLBACK statements are not valid in an IMS environment.

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original state without interfering with other system activities. Sometimes DB2 data does not return to a consistent state immediately. DB2 does not process data in an indoubt state until you restart IMS. To ensure that DB2 and IMS are synchronized, you must restart both DB2 and IMS.

Planning ahead for program recovery: Checkpoint and restart

Both IMS and DB2 handle recovery in an IMS application program that accesses DB2 data. IMS coordinates the process and DB2 participates by handling recovery for DB2 data.

There are two calls available to IMS programs to simplify program recovery: the *symbolic checkpoint* call and the *restart* call.

What symbolic checkpoint does
Symbolic checkpoint calls indicate to IMS that the program has reached a sync point. Such calls also establish places in the program from which you can restart the program.

A CHKP call causes IMS to:

- Inform DB2 that the changes your program made to the database can become permanent. DB2 makes the changes to DB2 data permanent, and IMS makes the changes to IMS data permanent.
- Send a message containing the checkpoint identification given in the call to the system console operator and to the IMS master terminal operator.
- Return the next input message to the program's I/O area if the program processes input messages. In MPPs and transaction-oriented BMPs, a checkpoint call acts like a call for a new message.
- Sign on to DB2 again, which resets special registers as follows:
  - CURRENT PACKAGESET to blanks
  - CURRENT SERVER to blanks
  - CURRENT SQLID to blanks
  - CURRENT DEGREE to 1

  Your program must restore those registers if their values are needed after the checkpoint.

Programs that issue symbolic checkpoint calls can specify as many as seven data areas in the program to be restored at restart. Symbolic checkpoint calls do not support OS/VS files; if your program accesses OS/VS files, you can convert those files to GSAM and use symbolic checkpoints. DB2 always recovers to the last checkpoint. You must restart the program from that point.

What restart does
The restart call (XRST), which you must use with symbolic checkpoints, provides a method for restarting a program after an abend. It restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.

When are checkpoints important?

Issuing checkpoint calls releases locked resources. The decision about whether or not your program should issue checkpoints (and if so, how often) depends on your program.

Generally, the following types of programs should issue checkpoint calls:
- Multiple-mode programs
- Batch-oriented BMPs
- Nonmessage-driven Fast Path programs (there is a special Fast Path call for these programs, but they can use symbolic checkpoint calls)
- Most batch programs
- Programs that run in a data sharing environment. (Data sharing makes it possible for online and batch application programs in separate IMS systems, in the same or separate processors, to access databases concurrently. Issuing checkpoint calls frequently in programs that run in a data sharing environment is important, because programs in several IMS systems access the database.)

You do not need to issue checkpoints in:
- Single-mode programs
- Database load programs
- Programs that access the database in read-only mode (defined with the processing option GO during a PSBGEN) and are short enough to restart from the beginning

- Programs that, by their nature, must have exclusive use of the database.

Checkpoints in MPPs and transaction-oriented BMPs

Single-mode programs: In single-mode programs, checkpoint calls and message retrieval calls (called get-unique calls) both establish commit points. The checkpoint calls retrieve input messages and take the place of get-unique calls. BMPs that access non-DL/I databases, and MPPs can issue both get unique calls and checkpoint calls to establish commit points.

However, message-driven BMPs must issue checkpoint calls rather than get-unique calls to establish commit points, because they can restart from a checkpoint only. If a program abends after issuing a get-unique call, IMS backs out the database updates to the most recent commit point—the get-unique call.

Multiple-mode programs: In multiple-mode BMPs and MPPs, the only commit points are the checkpoint calls that the program issues and normal program termination. If the program abends and it has not issued checkpoint calls, IMS backs out the program's database updates and cancels the messages it has created since the beginning of the program. If the program has issued checkpoint calls, IMS backs out the program's changes and cancels the output messages it has created since the most recent checkpoint call.

There are three considerations in issuing checkpoint calls in multiple-mode programs:
- How long it takes to back out and recover that unit of processing.

  The program must issue checkpoints frequently enough to make the program easy to back out and recover.
- How long database resources are locked in DB2 and IMS.
- How you want the output messages grouped.

  Checkpoint calls establish how a multiple-mode program groups its output messages. Programs must issue checkpoints frequently enough to avoid building up too many output messages.

Checkpoints in batch-oriented BMPs

Issuing checkpoints in a batch-oriented BMP is important for several reasons:
- To commit changes to the database
- To establish places from which the program can be restarted
- To release locked DB2 and IMS data that IMS has enqueued for the program.

Checkpoints also close all open cursors, which means you must reopen the cursors you want and re-establish positioning.

If a batch-oriented BMP does not issue checkpoints frequently enough, IMS can abend that BMP or another application program for one of these reasons:
- If a BMP retrieves and updates many database records between checkpoint calls, it can monopolize large portions of the databases and cause long waits for other programs needing those segments. (The exception to this is a BMP with a processing option of GO. IMS does not enqueue segments for programs with this processing option.) Issuing checkpoint calls releases the segments that the BMP has enqueued and makes them available to other programs.
- If IMS is using program isolation enqueuing, the space needed to enqueue information about the segments that the program has read and updated must not exceed the amount defined for the IMS system. If a BMP enqueues too many

segments, the amount of storage needed for the enqueued segments can exceed the amount of storage available. If that happens, IMS terminates the program abnormally with an abend code of U0775. You then have to increase the program's checkpoint frequency before rerunning the program. The amount of storage available is specified during IMS system definition. For more information, see *IMS Install Volume 2: System Definition and Tailoring*.

When you issue a DL/I CHKP call from an application program using DB2 databases, IMS processes the CHKP call for all DL/I databases, and DB2 commits all the DB2 database resources. No checkpoint information is recorded for DB2 databases in the IMS log or the DB2 log. The application program must record relevant information about DB2 databases for a checkpoint, if necessary.

One way to do this is to put such information in a data area included in the DL/I CHKP call. There can be undesirable performance implications of re-establishing position within a DB2 database as a result of the commit processing that takes place because of a DL/I CHKP call. The fastest way to re-establish a position in a DB2 database is to use an index on the target table, with a key that matches one-to-one with every column in the SQL predicate.

Another limitation of processing DB2 databases in a BMP program is that you can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.

Specifying checkpoint frequency

You must specify checkpoint frequency in your program in a way that makes it easy to change in case the frequency you initially specify is not right. Some ways to do this are:

- Use a counter in your program to keep track of elapsed time and issue a checkpoint call after a certain time interval.
- Use a counter to keep track of the number of root segments your program accesses. Issue a checkpoint call after a certain number of root segments.
- Use a counter to keep track of the number of updates your program performs. Issue a checkpoint call after a certain number of updates.

Unit of work in DL/I batch and IMS batch

This section describes how to coordinate commit and rollback operations for DL/I batch, and how to restart and recover in IMS batch.

Commit and rollback coordination

DB2 coordinates commit and rollback for DL/I batch, with the following considerations:

- DB2 and DL/I changes are committed as the result of IMS CHKP calls. However, you lose the application program database positioning in DL/I. In addition, the program database positioning in DB2 can be affected as follows:
  - If you did not specify the WITH HOLD option for a cursor, then you lose the position of that cursor.
  - If you specified the WITH HOLD option for a cursor and the application is message-driven, then you lose the position of that cursor.
  - If you specified the WITH HOLD option for a cursor and the application is operating in DL/I batch or DL/I BMP, then you retain the position of that cursor.

- DB2 automatically backs out changes whenever the application program abends. To back out DL/I changes, you must use the DL/I batch backout utility.
- You cannot use SQL statements COMMIT and ROLLBACK in the DB2 DL/I batch support environment, because IMS coordinates the unit of work. Issuing COMMIT causes SQLCODE -925 (SQLSTATE '2D521'); issuing ROLLBACK causes SQLCODE -926 (SQLSTATE '2D521').
- If the system fails, a unit of work resolves automatically when DB2 and IMS batch programs reconnect. If there is an indoubt unit of work, it resolves at reconnect time.
- You can use IMS rollback calls, ROLL and ROLB, to back out DB2 and DL/I changes to the last commit point. When you issue a ROLL call, DL/I terminates your program with an abend. When you issue a ROLB call, DL/I returns control to your program after the call.

  How ROLL and ROLB affect DL/I changes in a batch environment depends on the IMS system log used and the back out options specified, as the following summary indicates:

  - A ROLL call with tape logging (BKO specification does not matter), or disk logging and BKO=NO specified. DL/I does not back out updates and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
  - A ROLB call with tape logging (BKO specification does not matter), or disk logging and BKO=NO specified. DL/I does not back out updates and an AL status code returns in the PCB. DB2 backs out updates to the previous checkpoint. The DB2 DL/I support causes the application program to abend when ROLB fails.
  - A ROLL call with disk logging and BKO=YES specified. DL/I backs out updates and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
  - A ROLB call with disk logging and BKO=YES specified. DL/I backs out databases and control passes back to the application program. DB2 backs out updates to the previous checkpoint.

Using ROLL
Issuing a ROLL call causes IMS to terminate the program with a user abend code U0778. This terminates the program without a storage dump.

When you issue a ROLL call, the only option you supply is the call function, ROLL.

Using ROLB
The advantage of using ROLB is that IMS returns control to the program after executing ROLB, thus the program can continue processing. The options for ROLB are:
- The call function, ROLB
- The name of the I/O PCB.

In batch programs
If your IMS system log is on direct access storage, and if the run option BKO is Y to specify dynamic back out, you can use the ROLB call in a batch program. The ROLB call backs out the database updates since the last commit point and returns control to your program. You cannot specify the address of an I/O area as one of the options on the call; if you do, your program receives an AD status code. You must, however, have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB. For more information on using the CMPAT keyword, see *IMS Utilities Reference: System*.

Restart and recovery in IMS (batch)

In an online IMS system, recovery and restart are part of the IMS system. For a batch region, your location's operational procedures control recovery and restart. For more information, refer to *IMS Application Programming: Design Guide*.

Using savepoints to undo selected changes within a unit of work

Savepoints let you undo selected changes within a transaction. Your application can set any number of savepoints using SQL SAVEPOINT statements, and then use SQL ROLLBACK TO SAVEPOINT statements to indicate which changes within the unit of work to undo. When the application no longer uses a savepoint, it can delete that savepoint using the SQL RELEASE SAVEPOINT statement.

You can write a ROLLBACK TO SAVEPOINT statement with or without a savepoint name. If you do not specify a savepoint name, DB2 rolls back work to the most recently created savepoint.

Example: Rolling back to the most recently created savepoint: When the ROLLBACK TO SAVEPOINT statement is executed in the following code, DB2 rolls back work to savepoint B.

```
EXEC SQL SAVEPOINT A;
  .
  .
EXEC SQL SAVEPOINT B;
  .
  .
EXEC SQL ROLLBACK TO SAVEPOINT;
```

When savepoints are active, you cannot access remote sites using three-part names or aliases for three-part names. You can, however, use DRDA access with explicit CONNECT statements when savepoints are active. If you set a savepoint before you execute a CONNECT statement, the scope of that savepoint is the local site. If you set a savepoint after you execute the CONNECT statement, the scope of that savepoint is the site to which you are connected.

Example: Setting savepoints during distributed processing: Suppose that an application performs these tasks:
1. Sets savepoint C1
2. Does some local processing
3. Executes a CONNECT statement to connect to a remote site
4. Sets savepoint C2

Because savepoint C1 is set before the application connects to a remote site, savepoint C1 is known only at the local site. However, because savepoint C2 is set after the application connects to the remote site, savepoint C2 is known only at the remote site.

You can set a savepoint with the same name multiple times within a unit of work. Each time that you set the savepoint, the new value of the savepoint replaces the old value.

Example: Setting a savepoint multiple times: Suppose that the following actions take place within a unit of work:
1. Application A sets savepoint S.
2. Application A calls stored procedure P.
3. Stored procedure P sets savepoint S.
4. Stored procedure P executes ROLLBACK TO SAVEPOINT S.

When DB2 executes ROLLBACK to SAVEPOINT S, DB2 rolls back work to the savepoint that was set in the stored procedure because that value is the most recent value of savepoint S.

If you do not want a savepoint to have different values within a unit of work, you can use the UNIQUE option in the SAVEPOINT statement. If an application executes a SAVEPOINT statement for a savepoint that was previously defined as unique, an SQL error occurs.

Savepoints are automatically released at the end of a unit of work. However, if you no longer need a savepoint before the end of a transaction, you should execute the SQL RELEASE SAVEPOINT statement. Releasing savepoints is essential if you need to use three-part names to access remote locations.

Restrictions on using savepoints: You cannot use savepoints in global transactions, triggers, or user-defined functions, or in stored procedures, user-defined functions, or triggers that are nested within triggers or user-defined functions.

For more information on the SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT statements, see Chapter 5 of *DB2 SQL Reference*.

Chapter 19. Planning to access distributed data

An instance of DB2 for OS/390 and z/OS can communicate with other instances of the same product and with some other products. This chapter:

1. Introduces some background material, in "Introduction to accessing distributed data". A key point is that there are two methods of access that you ought to consider.
2. Tells how to design programs to for distributed access, using a sample task as illustration, in "Coding for distributed data by two methods" on page 369.
3. Discusses some considerations for choosing an access method, in "Coding considerations for access methods" on page 372.
4. Tells how to prepare programs that use the one method that requires special preparation, in "Preparing programs For DRDA access" on page 374.
5. Describes special considerations for a possibly complex situation, in "Coordinating updates to two or more data sources" on page 377.
6. Concludes with "Miscellaneous topics for distributed data" on page 378.

Introduction to accessing distributed data

Definitions: *Distributed data* is data that resides on some database management system (DBMS) other than your local system. Your *local* DBMS is the one on which you bind your application plan. All other DBMSs are *remote*.

In this chapter, we assume that you are requesting services from a remote DBMS. That DBMS is a *server* in that situation, and your local system is a *requester* or *client*.

Your application can be connected to many DBMSs at one time; the one currently performing work is the *current server*. When the local system is performing work, it also is called the current server.

A remote server can be truly remote in the physical sense: thousands of miles away. But that is not necessary; it could even be another subsystem of the same operating system your local DBMS runs under. We assume that your local DBMS is an instance of DB2 for OS/390 and z/OS. A remote server could be an instance of DB2 for OS/390 and z/OS also, or an instance of one of many other products.

A DBMS, whether local or remote, is known to your DB2 system by its *location name*. The location name of a remote DBMS is recorded in the communications database. (If you need more information about location names or the communications database, see Part 3 of *DB2 Installation Guide*.)

Example 1: You can write a query like this to access data at a remote server:

```
SELECT * FROM CHICAGO.DSN8710.EMP
  WHERE EMPNO = '0001000';
```

The mode of access depends on whether you bind your DBRMs into packages and on the value of field DATABASE PROTOCOL in installation panel DSNTIP5 or the value of bind option DBPROTOCOL. Bind option DBPROTOCOL overrides the installation setting.

Example 2: You can also write statements like these to accomplish the same task:

```
EXEC SQL
  CONNECT TO CHICAGO;
SELECT * FROM DSN8710.EMP
  WHERE EMPNO = '0001000';
```

Before you can execute the query at location CHICAGO, you must bind a package
at the CHICAGO server. You must also bind a plan or package at the local site.

Example 3: You can call a *stored procedure*, which is a subroutine that can contain
many SQL statements. Your program executes this:

```
EXEC SQL
  CONNECT TO ATLANTA;

EXEC SQL
  CALL procedure_name (parameter_list);
```

The parameter list is a list of host variables that is passed to the stored procedure
and into which it returns the results of its execution. The stored procedure must
already exist at location ATLANTA.

Two methods of access: The examples above show two different methods for
accessing distributed data.

* Example 1 shows a statement that can be executed with **DB2 private protocol
  access** or **DRDA access**.

  If you bind the DBRM that contains the statement into a plan at the local DB2
  and specify the bind option DBPROTOCOL(PRIVATE), you access the server
  using DB2 private protocol access.

  If you bind the DBRM that contains the statement using one of these methods,
  you access the server using DRDA access.

  Method 1:
  – Bind the DBRM into a package at the local DB2 using the bind option
    DBPROTOCOL(DRDA®).
  – Bind the DBRM into a package at the remote location (CHICAGO).
  – Bind the packages into a plan using bind option DBPROTOCOL(DRDA).

  Method 2:
  – Bind the DBRM into a package at the remote location.
  – Bind the remote package and the DBRM into a plan at the local site, using the
    bind option DBPROTOCOL(DRDA).

* Examples 2 and 3 show statements that are executed with **DRDA access** only.
  When you use these methods for DRDA access, your application must include an
  explicit CONNECT statement to switch your connection from one system to
  another.

Planning considerations for choosing an access method: DB2 private protocol
access and DRDA access differ in several ways. To choose between them, you
must know:

* **What kind of server you are querying.**

  DB2 private protocol access is available only to supported releases of DB2 for
  OS/390 and z/OS.

  DRDA access is available to all DBMSs that implement Distributed Relational
  Database Architecture (DRDA). Those include supported releases of DB2 for
  OS/390 and z/OS, other members of the DB2 family of IBM products, and many
  products of other companies.

* **What operations the server must perform.**

DB2 private protocol access supports only data manipulation statements: INSERT, UPDATE, DELETE, SELECT, OPEN, FETCH, and CLOSE. You cannot invoke user-defined functions and stored procedures or use LOBs or distinct types in applications that use DB2 private protocol access.

DRDA access allows any statement that the server can execute.

- **What performance you expect.**

  DRDA access has some significant advantages over DB2 private protocol access:

  - DRDA access uses a more compact format for sending data over the network, which improves the performance on slow network links.

  - Queries sent by DB2 private protocol access are bound at the server whenever they are first executed in a unit of work. Repeated binds can reduce the performance of a query that is executed often.

    A DBRM for statements executed by DRDA access is bound to a package at the server once. Those statements can include PREPARE and EXECUTE, so your application can accept dynamic statements to be executed at the server. But binding the package is an extra step in program preparation.

  - You can use stored procedures with DRDA access.

    While a stored procedure is running, it requires no message traffic over the network. That reduces the biggest hindrance to high performance for distributed data.

Recommendation: Use DRDA access whenever possible.

Other planning considerations: Authorization to connect to a remote server and to use resources there must be granted at the server to the appropriate authorization ID. For information when the server is DB2 for OS/390 and z/OS, see Part 3 (Volume 1) of *DB2 Administration Guide*. For information about other servers, see the documentation for the appropriate product.

If you update two or more DBMSs you must consider how updates can be coordinated, so that units of work at the two DBMSs are either both committed or both rolled back. Be sure to read "Coordinating updates to two or more data sources" on page 377.

You can use the resource limit facility at the server to govern distributed SQL statements. Governing is by plan for DB2 private protocol access and by package for DRDA access. See "Considerations for moving from DB2 private protocol access to DRDA access" on page 391 for information on changes you need to make to your resource limit facility tables when you move from DB2 private protocol access to DRDA access.

Coding for distributed data by two methods

This section illustrates the two ways to code applications for distributed access by the following hypothetical application:

Spiffy Computer has a master project table that supplies information about all projects currently active throughout the company. Spiffy has several branches in various locations around the world, each a DB2 location maintaining a copy of the project table named DSN8710.PROJ. The main branch location occasionally inserts data into all copies of the table. The application that makes the inserts uses a table of location names. For each row inserted, the application executes an INSERT statement in DSN8710.PROJ for each location.

Using three-part table names

You can use three-part table names to access data at a remote location through DRDA access or DB2 private protocol access. When you use three-part table names, the way you code your application is the same, regardless of the access method you choose. You determine the access method when you bind the SQL statements into a package or plan. If you use DRDA access, you must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server.

Because platforms other than DB2 for OS/390 and z/OS might not support the three-part name syntax, you should not code applications with three-part names if you plan to port those applications to other platforms.

In a three-part table name, the first part denotes the location. The local DB2 makes and breaks an implicit connection to a remote server as needed.

Spiffy's application uses a location name to construct a three-part table name in an INSERT statement. It then prepares the statement and executes it dynamically. (See "Chapter 23. Coding dynamic SQL in application programs" on page 495 for the technique.) The values to be inserted are transmitted to the remote location and substituted for the parameter markers in the INSERT statement.

The following overview shows how the application uses three-part names:

```
Read input values
Do for all locations
    Read location name
    Set up statement to prepare
    Prepare statement
    Execute statement
End loop
Commit
```

After the application obtains a location name, for example 'SAN_JOSE', it next creates the following character string:

```
INSERT INTO SAN_JOSE.DSN8710.PROJ VALUES (?,?,?,?,?,?,?,?)
```

The application assigns the character string to the variable INSERTX and then executes these statements:

```
EXEC SQL
    PREPARE STMT1 FROM :INSERTX;

EXEC SQL
    EXECUTE STMT1 USING :PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                        :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ;
```

The host variables for Spiffy's project table match the declaration for the sample project table in "Project table (DSN8710.PROJ)" on page 818.

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

Programming hint: You might find it convenient to use aliases when creating character strings that become prepared statements, instead of using full three-part

names like SAN_JOSE.DSN8710.PROJ. For information on aliases, see the section on CREATE ALIAS in *DB2 SQL Reference*.

Using explicit CONNECT statements

With this method the application program explicitly connects to each new server. You must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server.

In this example, Spiffy's application executes CONNECT for each server in turn and the server executes INSERT. In this case the tables to be updated each have the same name, though each is defined at a different server. The application executes the statements in a loop, with one iteration for each server.

The application connects to each new server by means of a host variable in the CONNECT statement. CONNECT changes the special register CURRENT SERVER to show the location of the new server. The values to insert in the table are transmitted to a location as input host variables.

The following overview shows how the application uses explicit CONNECTs:

```
Read input values
Do for all locations
    Read location name
    Connect to location
    Execute insert statement
End loop
Commit
Release all
```

The application inserts a new location name into the variable LOCATION_NAME, and executes the following statements:

```
EXEC SQL
   CONNECT TO :LOCATION_NAME;

EXEC SQL
   INSERT INTO DSN8710.PROJ VALUES (:PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                                    :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ);
```

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

The host variables for Spiffy's project table match the declaration for the sample project table in "Project table (DSN8710.PROJ)" on page 818. LOCATION_NAME is a character-string variable of length 16.

Releasing connections

When you connect to remote locations explicitly, you must also break those connections explicitly. You have considerable flexibility in determining how long connections remain open, so the RELEASE statement differs significantly from CONNECT.

Differences between CONNECT and RELEASE:

- CONNECT makes an immediate connection to exactly one remote system. CONNECT (Type 2) does not release any current connection.
- RELEASE

– *Does not* immediately break a connection. *The RELEASE statement labels connections for release at the next commit point.* A connection so labeled is in the *release-pending state* and can still be used before the next commit point.

– Can specify a single connection or a set of connections for release at the next commit point. The examples that follow show some of the possibilities.

Examples: Using the RELEASE statement, you can place any of the following in the release-pending state.

- A specific connection that the next unit of work does not use:

    EXEC SQL RELEASE SPIFFY1;

- The current SQL connection, whatever its location name:

    EXEC SQL RELEASE CURRENT;

- All connections except the local connection:

    EXEC SQL RELEASE ALL;

- All DB2 private protocol connections. If the first phase of your application program uses DB2 private protocol access and the second phase uses DRDA access, then open DB2 private protocol connections from the first phase could cause a CONNECT operation to fail in the second phase. To prevent that error, execute the following statement before the commit operation that separates the two phases:

    EXEC SQL RELEASE ALL PRIVATE;

    PRIVATE refers to DB2 private protocol connections, which exist only between instances of DB2 for OS/390 and z/OS.

Coding considerations for access methods

Stored procedures: If you use DRDA access, your program can call stored procedures at other systems that support them. Stored procedures behave like subroutines that can contain SQL statements as well as other operations. Read about them in "Chapter 24. Using stored procedures for client/server processing" on page 525.

SQL Limitations at Dissimilar Servers: Generally, a program using DRDA access can use SQL statements and clauses that are supported by a remote server even if they are not supported by the local server. *DB2 SQL Reference* tells what DB2 for OS/390 and z/OS supports; similar documentation is usually available for other products. The following examples suggest what to expect from dissimilar servers:

- They support SELECT, INSERT, UPDATE, DELETE, DECLARE CURSOR, and FETCH, but details vary.

    Example: SQL/DS™ does not support the clause WITH HOLD on DECLARE CURSOR.

- Data definition statements vary more widely.

    Example: SQL/DS does not support CREATE DATABASE. It does support ACQUIRE DBSPACE for a similar purpose.

- Statements can have different limits.

    Example: A query in DB2 for OS/390 and z/OS can have 750 columns; for another system, the maximum might be 255. But a query using 255 or fewer columns could execute in both systems.

- Some statements are not sent to the server but are processed completely by the requester. You cannot use those statements in a remote package even though

the server supports them. For a list of those statements, see "Appendix G. Characteristics of SQL statements in DB2 for OS/390 and z/OS" on page 919.

- In general, if a statement to be executed at a remote server contains host variables, a DB2 requester assumes them to be input host variables unless it supports the syntax of the statement and can determine otherwise. If the assumption is not valid, the server rejects the statement.

Three-part names and multiple servers: If you use a three-part name, or an alias that resolves to one, in a statement executed at a remote server by DRDA access, and if the location name is not that of the server, then the method by which the remote server accesses data at the named location depends on the value of DBPROTOCOL. If the package at the first remote server is bound with DBPROTOCOL(PRIVATE), DB2 uses DB2 private protocol access to access the second remote server. If the package at the first remote server is bound with DBPROTOCOL(DRDA), DB2 uses DRDA access to access the second remote server. We recommend that you follow these steps so that access to the second remote server is by DRDA access:

- Rebind the package at the first remote server with DBPROTOCOL(DRDA).
- Bind the package that contains the three-part name at the second server.

Accessing declared temporary tables using three-part names: You can access a remote declared temporary table using a three-part name only if you use DRDA access. However, if you combine explicit CONNECT statements and three-part names in your application, a reference to a remote declared temporary table must be a forward reference. For example, you can perform the following series of actions, which includes a forward reference to a declared temporary table:

```
EXEC SQL CONNECT TO CHICAGO;              /* Connect to the remote site */
EXEC SQL
  DECLARE GLOBAL TEMPORARY TABLE TEMPPROD /* Define the temporary table */
  (CHARCOL CHAR(6) NOT NULL);             /* at the remote site          */
EXEC SQL CONNECT RESET;                   /* Connect back to local site */
EXEC SQL INSERT INTO CHICAGO.SESSION.T1
  (VALUES 'ABCDEF');                      /* Access the temporary table*/
                          /* at the remote site (forward reference) */
```

However, you cannot perform the following series of actions, which includes a backward reference to the declared temporary table:

```
EXEC SQL
  DECLARE GLOBAL TEMPORARY TABLE TEMPPROD /* Define the temporary table */
  (CHARCOL CHAR(6) NOT NULL);             /* at the local site (ATLANTA)*/
EXEC SQL CONNECT TO CHICAGO;              /* Connect to the remote site */
EXEC SQL INSERT INTO ATLANTA.SESSION.T1
  (VALUES 'ABCDEF');                      /* Cannot access temp table  */
                          /* from the remote site (backward reference)*/
```

Savepoints: In a distributed environment, you can set savepoints only if you use DRDA access with explicit CONNECT statements. If you set a savepoint and then execute an SQL statement with a three-part name, an SQL error occurs.

The site at which a savepoint is recognized depends on whether the CONNECT statement is executed before or after the savepoint is set. For example, if an application executes the statement SET SAVEPOINT C1 at the local site before it executes a CONNECT TO S1 statement, savepoint C1 is known only at the local site. If the application executes CONNECT to S1 before SET SAVEPOINT C1, the savepoint is known only at site S1.

For more information on savepoints, see "Using savepoints to undo selected changes within a unit of work" on page 365.

Scrollable cursors: In a distributed environment, you can use scrollable cursors only if you use DRDA access.

Preparing programs For DRDA access

For the most part, binding a package to run at a remote location is like binding a package to run at your local DB2. Binding a plan to run the package is like binding any other plan. For the general instructions, see "Chapter 20. Preparing an application program to run" on page 395. This section describes the few differences.

Precompiler options

The following precompiler options are relevant to preparing a package to be run using DRDA access:

CONNECT
Use **CONNECT(2)**, explicitly or by default.

CONNECT(1) causes your CONNECT statements to allow only the restricted function known as "remote unit of work". Be particularly careful to avoid CONNECT(1) if your application updates more than one DBMS in a single unit of work.

SQL
Use **SQL(ALL)** explicitly for a package that runs on a server that *is not* DB2 for OS/390 and z/OS. The precompiler then accepts any statement that obeys DRDA rules.

Use **SQL(DB2)**, explicitly or by default, if the server is DB2 for OS/390 and z/OS only. The precompiler then rejects any statement that does not obey the rules of DB2 for OS/390 and z/OS.

BIND PACKAGE options

The following options of BIND PACKAGE are relevant to binding a package to be run using DRDA access:

location-name
Name the location of the server at which the package runs.

The privileges needed to run the package must be granted to the owner of the package at the server. If you are not the owner, you must also have SYSCTRL authority or the BINDAGENT privilege granted locally.

SQLERROR
Use **SQLERROR(CONTINUE)** if you used SQL(ALL) when precompiling. That creates a package even if the bind process finds SQL errors, such as statements that are valid on the remote server but that the precompiler did not recognize. Otherwise, use SQLERROR(NOPACKAGE), explicitly or by default.

CURRENTDATA
Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors. See "Use block fetch" on page 383 for more information.

OPTIONS
When you make a remote copy of a package using BIND PACKAGE with the COPY option, use this option to control the default bind options that DB2 uses. Specify:

COMPOSITE to cause DB2 to use any options you specify in the BIND PACKAGE command. For all other options, DB2 uses the options of the copied package. This is the default.
COMMAND to cause DB2 to use the options you specify in the BIND PACKAGE command. For all other options, DB2 uses the defaults for the server on which the package is bound. This helps ensure that the server supports the options with which the package is bound.

DBPROTOCOL

Use **DBPROTOCOL(PRIVATE)** if you want DB2 to use DB2 private protocol access for accessing remote data that is specified with three-part names.

Use **DBPROTOCOL(DRDA)** if you want DB2 to use DRDA access to access remote data that is specified with three-part names. You must bind a package at all locations whose names are specified in three-part names.

These values override the value of DATABASE PROTOCOL on installation panel DSNTIP5. Therefore, if the setting of DATABASE PROTOCOL at the requester site specifies the type of remote access you want to use for three-part names, you do not need to specify the DBPROTOCOL bind option.

ENCODING

Use this option to control the encoding scheme that is used for static SQL statements in the package and to set the initial value of the CURRENT APPLICATION ENCODING SCHEME special register.

The default ENCODING value for a package that is bound at a remote DB2 for OS/390 and z/OS server is the system default for that server. The system default is specified at installation time in the APPLICATION ENCODING field of panel DSNTIPF.

For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the *plan*. For applications that execute remotely and use implicit CONNECT statements, DB2 uses the ENCODING value for the *package* that is at the site where a statement executes.

BIND PLAN options

The following options of BIND PLAN are particularly relevant to binding a plan that uses DRDA access:

DISCONNECT

For most flexibility, use DISCONNECT(EXPLICIT), explicitly or by default. That requires you to use RELEASE statements in your program to explicitly end connections.

But the other values of the option are also useful.
DISCONNECT(AUTOMATIC) ends all remote connections during a commit operation, without the need for RELEASE statements in your program.
DISCONNECT(CONDITIONAL) ends remote connections during a commit operation except when an open cursor defined as WITH HOLD is associated with the connection.

SQLRULES

Use **SQLRULES(DB2)**, explicitly or by default.

SQLRULES(STD) applies the rules of the SQL standard to your CONNECT statements, so that CONNECT TO *x* is an error if you are already connected to *x*. Use STD only if you want that statement to return an error code.

If your program selects LOB data from a remote location, and you bind the plan for the program with SQLRULES(DB2), the format in which you retrieve the

LOB data with a cursor is restricted. After you open the cursor to retrieve the LOB data, you must retrieve all of the data using a LOB variable, or retrieve all of the data using a LOB locator variable. If the value of SQLRULES is STD, this restriction does not exist.

If you intend to switch between LOB variables and LOB locators to retrieve data from a cursor, execute the SET SQLRULES=STD statement before you connect to the remote location.

CURRENTDATA
Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors. See "Use block fetch" on page 383 for more information.

DBPROTOCOL
Use **DBPROTOCOL(PRIVATE)** if you want DB2 to use DB2 private protocol access for accessing remote data that is specified with three-part names.

Use **DBPROTOCOL(DRDA)** if you want DB2 to use DRDA access to access remote data that is specified with three-part names. You must bind a package at all locations whose names are specified in three-part names.

The package value for the DBPROTOCOL option overrides the plan option. For example, if you specify DBPROTOCOL(DRDA) for a remote package and DBPROTOCOL(PRIVATE) for the plan, DB2 uses DRDA access when it accesses data at that location using a three-part name. If you do not specify any value for DBPROTOCOL, DB2 uses the value of DATABASE PROTOCOL on installation panel DSNTIP5.

ENCODING
Use this option to control the encoding scheme that is used for static SQL statements in the plan and to set the initial value of the CURRENT APPLICATION ENCODING SCHEME special register.

For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. For applications that execute remotely and use implicit CONNECT statements, DB2 uses the ENCODING value for the *package* that is at the site where a statement executes.

Checking BIND PACKAGE options

You can request only the options of BIND PACKAGE that are supported by the server. But you must specify those options at the requester using the requester's syntax for BIND PACKAGE. To find out which options are supported by a specific server DBMS, check the documentation provided for that server. If the server recognizes an option by a different name, the table of generic descriptions in "Appendix H. Program preparation options for remote packages" on page 929 might help to identify it.

- Guidance in using DB2 bind options and performing a bind process is documented in this book, especially in "Chapter 20. Preparing an application program to run" on page 395.
- For the syntax of DB2 BIND and REBIND subcommands, see Chapter 2 of *DB2 Command Reference*.
- For a list of DB2 bind options in generic terms, including options you cannot request from DB2 but can use if you request from a non-DB2 server, see "Appendix H. Program preparation options for remote packages" on page 929.

Coordinating updates to two or more data sources

Definition: Two or more updates are *coordinated* if they must all commit or all roll back in the same unit of work.

Updates to two or more DBMSs can be coordinated automatically if both systems implement a method called *two-phase commit*.

Example: The situation is common in banking: an amount is subtracted from one account and added to another. The two actions must either both commit or both roll back at the end of the unit of work.

DB2 and IMS, and DB2 and CICS, jointly implement a two-phase commit process. You can update an IMS database and a DB2 table in the same unit of work. If a system or communication failure occurs between committing the work on IMS and on DB2, then the two programs restore the two systems to a consistent point when activity resumes.

Details of the two-phase commit process are not important to the rest of this description. You can read them in Part 4 (Volume 1) of *DB2 Administration Guide*.

How to have coordinated updates

Ideally, work only with systems that implement two-phase commit.

Versions 3 and later of DB2 for OS/390 and z/OS implement two-phase commit. For other types of DBMS, check the product specifications.

Example: The examples described under "Using three-part table names" on page 370 and "Using explicit CONNECT statements" on page 371 assume that all systems involved implement two-phase commit. Both examples suggest updating several systems in a loop and ending the unit of work by committing only when the loop is over. In both cases, updates are coordinated across the entire set of systems.

Restrictions on updates at servers that Do Not Support Two-Phase Commit:
You cannot really have coordinated updates with a DBMS that does not implement two-phase commit. In the description that follows, we call such a DBMS a *restricted system*. DB2 prevents you from updating both a restricted system and also any other system in the same unit of work. In this context, *update* includes the statements INSERT, DELETE, UPDATE, CREATE, ALTER, DROP, GRANT, REVOKE, and RENAME.

To achieve the effect of coordinated updates with a restricted system, you must first update one system and commit that work, and then update the second system and commit its work. If a failure occurs after the first update is committed and before the second is committed, there is no automatic provision for bringing the two systems back to a consistent point. Your program must assume that task.

> **CICS and IMS**
> You cannot update at servers that do not support two-phase commit.

> **TSO and batch**
>
> You can update if and only if:
>
> - No other connections exist, or
> - All existing connections are to servers that are restricted to read-only operations.
>
> If these conditions are not met, then you are restricted to read-only operations.
>
> If the first connection in a logical unit of work is to a server that supports two-phase commit, and there are no existing connections or only read-only connections, then that server and all servers that support two-phase commit can update. However, if the first connection is to a server that does not support two-phase commit, only that server is allowed to update.

Recommendation: Rely on DB2 to prevent updates to two systems in the same unit of work if either of them is a restricted system.

What you can do without two-phase commit

If you are accessing a mixture of systems, some of which might be restricted, you can:

- Read from any of the systems at any time.
- Update any one system many times in one unit of work.
- Update many systems, including CICS or IMS, in one unit of work, provided that none of them is a restricted system. If the first system you update in a unit of work is not restricted, any attempt to update a restricted system in that unit of work returns an error.
- Update one restricted system in a unit of work, provided that you do not try to update any other system in the same unit of work. If the first system you update in a unit of work is restricted, any attempt to update any other system in that unit of work returns an error.

Restricting to CONNECT (type 1): You can also restrict your program completely to the rules for restricted systems, by using the type 1 rules for CONNECT. Those rules are compatible with packages that were bound on Version 2 Release 3 of DB2 for MVS and were not rebound on a later version. To put those rules into effect for a package, use the precompiler option CONNECT(1). Be careful not to use packages precompiled with CONNECT(1) and packages precompiled with CONNECT(2) in the same package list. The first CONNECT statement executed by your program determines which rules are in effect for the entire execution: type 1 or type 2. An attempt to execute a later CONNECT statement precompiled with the other type returns an error.

For more information about CONNECT (Type 1) and about managing connections to other systems, see Chapter 1 of *DB2 SQL Reference*.

Miscellaneous topics for distributed data

Selecting an access method and managing connections to other systems are the critical elements in designing a program to use distributed data. This section contains advice about other topics:

- "Improving performance for remote access" on page 379

- "Maximizing LOB performance in a distributed environment"
- "Specifying OPTIMIZE FOR n ROWS" on page 385
- "Specifying FETCH FIRST n ROWS ONLY" on page 388
- "Maintaining data currency" on page 390
- "Copying a table from a remote location" on page 390
- "Transmitting mixed data" on page 390
- "Considerations for moving from DB2 private protocol access to DRDA access" on page 391

Improving performance for remote access

A query sent to a remote subsystem almost always takes longer to execute than the same query that accesses tables of the same size on the local subsystem. The principle causes are:

- Overhead processing, including start up, negotiating session limits, and, for DB2 private protocol access, the bind required at the remote location
- The time required to send messages across the network.

Code efficient queries

To gain the greatest efficiency when accessing remote subsystems, compared to that on similar tables at the local subsystem, try to write queries that send few messages over the network. To achieve that, try to:

- Reduce the number of columns and rows in the result table that is sent back to your application. Keep your SELECT lists as short as possible. Use the clauses WHERE, GROUP BY, and HAVING creatively, to eliminate unwanted data at the remote server.

- Use FOR FETCH ONLY or FOR READ ONLY. For example, retrieving thousands of rows as a continuous stream is reasonable. Sending a separate message for each one can be significantly slower.

- When possible, do not bind application plans and packages with ISOLATION(RR), even though that is the default. If your application does not need to refer again to rows it has once read, another isolation level could reduce lock contention and message overhead during COMMIT processing.

- Minimize the use of parameter markers.

  When your program uses DRDA access, DB2 can streamline the processing of dynamic queries that do not have parameter markers.

  When a DB2 requester encounters a PREPARE statement for such a query, it anticipates that the application is going to open a cursor. The requester therefore sends a single message to the server that contains a combined request for PREPARE, DESCRIBE, and OPEN. A DB2 server that receives such a message returns a single reply message that includes the output from the PREPARE, DESCRIBE, and OPEN operations. Thus, the number of network messages sent and received for these operations is reduced from 2 to 1.

  DB2 combines messages for these queries regardless of whether the bind option DEFER(PREPARE) is specified.

Maximizing LOB performance in a distributed environment

If you use DRDA access, you can access LOB columns in a remote table. Because LOB values are usually quite large, you need to use techniques for data retrieval that minimize the number of bytes transferred between the client and server.

Use LOB locators instead of LOB host variables: If you need to store only a portion of a LOB value at the client, or if your client program manipulates the LOB data but does not need a copy of it, LOB locators are a good choice. When a client

program retrieves a LOB column from a server into a locator, DB2 transfers only the 4 byte locator value to the client, not the entire LOB value. For information on how to use LOB locators in an application, see "Using LOB locators to save storage" on page 236.

Use stored procedure result sets: When you return LOB data to a client program from a stored procedure, use result sets, rather than passing the LOB data to the client in parameters. Using result sets to return data causes less LOB materialization and less movement of data among address spaces. For information on how to write a stored procedure to return result sets, see "Writing a stored procedure to return result sets to a DRDA client" on page 545. For information on how to write a client program to receive result sets, see "Writing a DB2 for OS/390 and z/OS client program or SQL procedure to receive result sets" on page 599.

Set the CURRENT RULES special register to DB2: When a DB2 for OS/390 and z/OS server receives an OPEN request for a cursor, the server uses the value in the CURRENT RULES special register to determine the type of host variables the associated statement uses to retrieve LOB values. If you specify a value of DB2 for CURRENT RULES before you perform a CONNECT, and the first FETCH for the cursor uses a LOB locator to retrieve LOB column values, DB2 lets you use only LOB locators for all subsequent FETCH statements for that column until you close the cursor. If the first FETCH uses a host variable, DB2 lets you use only host variables for all subsequent FETCH statements for that column until you close the cursor. However, if you set the value of CURRENT RULES to STD, DB2 lets you use the same open cursor to fetch a LOB column into either a LOB locator or a host variable.

Although a value of STD for CURRENT RULES gives you more programming flexibility when you retrieve LOB data, you get better performance if you use a value of DB2. With the STD option, the server must send and receive network messages for each FETCH to indicate whether the data being transferred is a LOB locator or a LOB value. With the DB2 option, the server knows the size of the LOB data after the first FETCH, so an extra message about LOB data size is unnecessary. The server can send multiple blocks of data to the requester at one time, which reduces the total time for data transfer.

For example, an end user might want to browse through a large set of employee records but want to look at pictures of only a few of those employees. At the server, you set the CURRENT RULES special register to DB2. In the application, you declare and open a cursor to select employee records. The application then fetches all picture data into 4 byte LOB locators. Because DB2 knows that 4 bytes of LOB data is returned for each FETCH, DB2 can fill the network buffers with locators for many pictures. When a user wants to see a picture for a particular person, the application can retrieve the picture from the server by assigning the value referenced by the LOB locator to a LOB host variable:

```
SQL TYPE IS BLOB my_blob[1M];
SQL TYPE IS BLOB AS LOCATOR my_loc;
  :
  :
FETCH C1 INTO :my_loc;     /* Fetch BLOB into LOB locator  */
  :
  :
SET :my_blob = :my_loc; /* Assign BLOB to host variable */
```

Use bind options that improve performance

Your choice of these bind options can affect the performance of your distributed applications:
- DEFER(PREPARE) or NODEFER(PREPARE)

- REOPT(VARS) or NOREOPT(VARS)
- CURRENTDATA(YES) or CURRENTDATA(NO)
- KEEPDYNAMIC(YES) or KEEPDYNAMIC(NO)
- DBPROTOCOL(PRIVATE) or DBPROTOCOL(DRDA)

DEFER(PREPARE)

To improve performance for both static and dynamic SQL used in DB2 private protocol access, and for dynamic SQL in DRDA access, consider specifying the option DEFER(PREPARE) when you bind or rebind your plans or packages. Remember that statically bound SQL statements in DB2 private protocol access are processed dynamically. When a dynamic SQL statement accesses remote data, the PREPARE and EXECUTE statements can be transmitted over the network together and processed at the remote location, and responses to both statements can be sent together back to the local subsystem, thus reducing traffic on the network. DB2 does not prepare the dynamic SQL statement until the statement executes. (The exception to this is dynamic SELECT, which combines PREPARE and DESCRIBE, whether or not the DEFER(PREPARE) option is in effect.)

All PREPARE messages for dynamic SQL statements that refer to a remote object will be deferred until either:
- The statement executes
- The application requests a description of the results of the statement.

In general, when you defer PREPARE, DB2 returns SQLCODE 0 from PREPARE statements. You must therefore code your application to handle any SQL codes that might have been returned from the PREPARE statement after the associated EXECUTE or DESCRIBE statement.

When you use predictive governing, the SQL code returned to the requester if the server exceeds a predictive governing warning threshold depends on the level of DRDA at the requester. See "Writing an application to handle predictive governing" on page 503 for more information.

For DB2 private protocol access, when a static SQL statement refers to a remote object, the transparent PREPARE statement and the EXECUTE statements are automatically combined and transmitted across the network together. The PREPARE statement is deferred only if you specify the bind option DEFER(PREPARE).

PREPARE statements that contain INTO clauses are not deferred.

PKLIST

The order in which you specify package collections in a package list can affect the performance of your application program. When a local instance of DB2 attempts to execute an SQL statement at a remote server, the local DB2 subsystem must determine which package collection the SQL statement is in. DB2 must send a message to the server, requesting that the server check each collection ID for the SQL statement, until the statement is found or there are no more collection IDs in the package list. You can reduce the amount of network traffic, and thereby improve performance, by reducing the number of package collections that each server must search. These examples show ways to reduce the collections to search:

- Reduce the number of packages per collection that must be searched. The following example specifies only 1 package in each collection:

  ```
  PKLIST(S1.COLLA.PGM1, S1.COLLB.PGM2)
  ```

- Reduce the number of package collections at each location that must be searched. The following example specifies only 1 package collection at each location:

  ```
  PKLIST(S1.COLLA.*, S2.COLLB.*)
  ```
- Reduce the number of collections used for each application. The following example specifies only 1 collection to search:

  ```
  PKLIST(*.COLLA.*)
  ```

You can also specify the package collection associated with an SQL statement in your application program. Execute the SQL statement SET CURRENT PACKAGESET before you execute an SQL statement to tell DB2 which package collection to search for the statement.

When you use DEFER(PREPARE) with DRDA access, the package containing the statements whose preparation you want to defer must be the *first* qualifying entry in DB2's package search sequence. (See "Identifying packages at run time" on page 412 for more information.) For example, assume that the package list for a plan contains two entries:

```
PKLIST(LOCB.COLLA.*, LOCB.COLLB.*)
```

If the intended package is in collection COLLB, ensure that DB2 searches that collection first. You can do this by executing the SQL statement

```
SET CURRENT PACKAGESET = 'COLLB';
```

or by listing COLLB first in the PKLIST parameter of BIND PLAN:

```
PKLIST(LOCB.COLLB.*, LOCB.COLLA.*)
```

For NODEFER(PREPARE), the collections in the package list can be in any order, but if the package is not found in the first qualifying PKLIST entry, there is significant network overhead for searching through the list.

REOPT(VARS)

When you specify REOPT(VARS), DB2 determines access paths at both bind time and run time for statements that contain one or more of the following variables:
- Host variables
- Parameter markers
- Special registers

At run time, DB2 uses the values in those variables to determine the access paths.

If you specify the bind option REOPT(VARS), DB2 sets the bind option DEFER(PREPARE) automatically.

Because there are performance costs when DB2 reoptimizes the access path at run time, we recommend that you do the following:
- Use the bind option REOPT(VARS) only on packages or plans that contain statements that perform poorly because of a bad access path.
- Use the option NOREOPT(VARS) when you bind a plan or package that contains statements that use DB2 private protocol access.

  If you specify REOPT(VARS) when you bind a plan that contains statements that use DB2 private protocol access to access remote data, DB2 prepares those statements twice. See "How bind option REOPT(VARS) affects dynamic SQL" on page 523 for more information on REOPT(VARS).

CURRENTDATA(NO)

Use this bind option to force block fetch for ambiguous queries. See "Use block fetch" for more information on block fetch.

KEEPDYNAMIC(YES)

Use this bind option to improve performance for queries that use cursors defined WITH HOLD. With KEEPDYNAMIC(YES), DB2 automatically closes the cursor when there is no more data to retrieve. The client does not need to send a network message to tell DB2 to close the cursor. For more information on KEEPDYNAMIC(YES), see "Keeping prepared statements after commit points" on page 500.

DBPROTOCOL(DRDA)

If the value of installation default DATABASE PROTOCOL is not DRDA, use this bind option to cause DB2 to use DRDA access to execute SQL statements with three-part names. Statements that use DRDA access perform better at execution time because:

* Binding occurs when the package is bound, not during program execution.
* DB2 does not destroy static statement information at COMMIT time, as it does with DB2 private protocol access. This means that with DRDA access, if a COMMIT occurs between two executions of a statement, DB2 does not need to prepare the statement twice.

Use block fetch

DB2 uses two different methods to reduce the number of messages sent across the network when fetching data using a cursor:

* *Limited block fetch* optimizes data transfer by guaranteeing the transfer of a minimum amount of data in response to each request from the requesting system.
* *Continuous block fetch* sends a single request from the requester to the server. The server fills a buffer with data it retrieves and transmits it back to the requester. Processing at the requester is asynchronous with the server; the server continues to send blocks of data to the requester with minimal or no further prompting.

See the information on block fetch in Part 5 (Volume 2) of *DB2 Administration Guide* for more information.

How to ensure block fetching: To use either type of block fetch, DB2 must determine that the cursor is not used for updating or deleting. Indicate that the cursor does not modify data by adding FOR FETCH ONLY or FOR READ ONLY to the query in the DECLARE CURSOR statement. If you do not use FOR FETCH ONLY or FOR READ ONLY, DB2 still uses block fetch for the query if:

* The cursor is a non-scrollable cursor, and the result table of the cursor is read-only. (See Chapter 5 of *DB2 SQL Reference* for a description of read-only tables.)
* The cursor is a scrollable cursor that is declared as INSENSITIVE, and the result table of the cursor is read-only.
* The cursor is a scrollable cursor that is declared as SENSITIVE, the result table of the cursor is read-only, and the value of bind option CURRENTDATA is NO.
* The result table of the cursor is not read-only, but the cursor is ambiguous, and the value of bind option CURRENTDATA is NO. A cursor is ambiguous when:
  - It is not defined with the clauses FOR FETCH ONLY, FOR READ ONLY, or FOR UPDATE OF.

- It is not defined on a read-only result table.
- It is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement.
- It is in a plan or package that contains the SQL statements PREPARE or EXECUTE IMMEDIATE.

DB2 does not use continuous block fetch if:

- The cursor is referred to in the statement DELETE WHERE CURRENT OF elsewhere in the program.
- The cursor statement appears that it can be updated at the requesting system. (DB2 does not check whether the cursor references a view at the server that cannot be updated.)

When DB2 uses block fetch for non-scrollable cursors

Table 45 summarizes the conditions under which a DB2 server uses block fetch for a non-scrollable cursor.

Table 45. Effect of CURRENTDATA and isolation level on block fetch for a non-scrollable cursor

Isolation	CURRENTDATA	Cursor Type	Block Fetch
CS, RR, or RS	YES	Read-only	Yes
		Updatable	No
		Ambiguous	No
	No	Read-only	Yes
		Updatable	No
		Ambiguous	Yes
UR	Yes	Read-only	Yes
	No	Read-only	Yes

When DB2 uses block fetch for scrollable cursors

Table 46 summarizes the conditions under which a DB2 server uses block fetch for a scrollable cursor when the cursor is not used to retrieve result sets.

Table 46. Effect of CURRENTDATA and isolation level on block fetch for a scrollable cursor that is not used for a stored procedure result set

Isolation	Cursor sensitivity	CURRENT–DATA	Cursor type	Block fetch
CS, RR, or RS	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	Yes
			Updatable	No
			Ambiguous	No
		No	Read-only	Yes
			Updatable	No
			Ambiguous	Yes

Table 46. Effect of CURRENTDATA and isolation level on block fetch for a scrollable cursor that is not used for a stored procedure result set (continued)

Isolation	Cursor sensitivity	CURRENT–DATA	Cursor type	Block fetch
UR	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes

Table 47 summarizes the conditions under which a DB2 server uses block fetch for a scrollable cursor when the cursor is used to retrieve result sets.

Table 47. Effect of CURRENTDATA and isolation level on block fetch for a scrollable cursor that is used for a stored procedure result set

Isolation	Cursor sensitivity	CURRENT–DATA	Cursor type	Block fetch
CS, RR, or RS	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	No
		No	Read-only	Yes
UR	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes

When a DB2 for OS/390 and z/OS requester uses a scrollable cursor to retrieve data from a DB2 for OS/390 and z/OS server, the following conditions are true:

- The requester never requests more than 64 rows in a query block, even if more rows fit in the query block. In addition, the requester never requests extra query blocks. This is true even if the setting of field EXTRA BLOCKS REQ in the DISTRIBUTED DATA FACILITY PANEL 2 installation panel on the requester allows extra query blocks to be requested. If you want to limit the number of rows that the server returns to fewer than 64, you can specify the FETCH FIRST *n* ROWS ONLY clause when you declare the cursor.

- The requester discards rows of the result table if the application does not use those rows. For example, if the application fetches row *n* and then fetches row *n*+2, the requester discards row *n*+1. The application gets better performance for a blocked scrollable cursor if it mostly scrolls forward, fetches most of the rows in a query block, and avoids frequent switching between FETCH ABSOLUTE statements with negative and positive values.

- If the scrollable cursor does not use block fetch, the server returns one row for each FETCH statement.

Specifying OPTIMIZE FOR n ROWS

You can use the clause OPTIMIZE FOR *n* ROWS in your SELECT statements to limit the number of data rows that the server returns on each DRDA network transmission. You can also use OPTIMIZE FOR *n* ROWS to return a query result set from a stored procedure. OPTIMIZE FOR *n* ROWS has no effect on scrollable cursors.

The number of rows that DB2 transmits on each network transmission depends on the following factors:

- If *n* rows of the SQL result set fit within a single DRDA query block, a DB2 server can send *n* rows to any DRDA client. In this case, DB2 sends *n* rows in each network transmission, until the entire query result set is exhausted.
- If *n* rows of the SQL result set exceed a single DRDA query block, the number of rows that are contained in each network transmission depends on the client's DRDA software level and configuration:
  - If the client does not support extra query blocks, the DB2 server automatically reduces the value of *n* to match the number of rows that fit within a DRDA query block.
  - If the client supports extra query blocks, the DRDA client can choose to accept multiple DRDA query blocks in a single data transmission. DRDA allows the client to establish an upper limit on the number of DRDA query blocks in each network transmission.

    The number of rows that a DB2 server sends is the smaller of *n* rows and the number of rows that fit within the lesser of these two limitations:
    - The value of EXTRA BLOCKS SRV in install panel DSNTIP5 at the DB2 server

      This is the maximum number of extra DRDA query blocks that the DB2 server returns to a client in a single network transmission.
    - The client's extra query block limit, which is obtained from the DDM MAXBLKEXT parameter received from the client

      When DB2 acts as a DRDA client, the DDM MAXBLKEXT parameter is set to the value that is specified on the EXTRA BLOCKS REQ install option of the DSNTIP5 install panel.

The OPTIMIZE FOR *n* ROWS clause is useful in two cases:

- If *n* is less than the number of rows that fit in the DRDA query block, OPTIMIZE FOR *n* ROWS can improve performance by preventing the DB2 server from fetching rows that might never be used by the DRDA client application.
- If *n* is greater than the number of rows that fit in a DRDA query block, OPTIMIZE FOR *n* ROWS lets the DRDA client request multiple blocks of query data on each network transmission. This use of OPTIMIZE FOR *n* ROWS can significantly improve elapsed time for large query download operations.

Specifying a large value for *n* in OPTIMIZE FOR *n* ROWS can increase the number of DRDA query blocks that a DB2 server returns in each network transmission. This function can improve performance significantly for applications that use DRDA access to download large amounts of data. However, this same function can degrade performance if you do not use it properly. The examples below demonstrate the performance problems that can occur when you do not use OPTIMIZE FOR *n* ROWS judiciously.

In Figure 123 on page 387, the DRDA client opens a cursor and fetches rows from the cursor. At some point before all rows in the query result set are returned, the application issues an SQL INSERT. DB2 uses normal DRDA blocking, which has two advantages over the blocking that is used for OPTIMIZE FOR *n* ROWS:

- If the application issues an SQL statement other than FETCH (the example shows an INSERT statement), the DRDA client can transmit the SQL statement immediately, because the DRDA connection is not in use after the SQL OPEN.
- If the SQL application closes the cursor before fetching all the rows in the query result set, the server fetches only the number of rows that fit in one query block,

which is 100 rows of the result set. Basically, the DRDA query block size places an upper limit on the number of rows that are fetched unnecessarily.

DRDA client   DB2 server

```
DECLARE C1 CURSOR
    FOR SELECT * FROM T1
    FOR FETCH ONLY;

OPEN C1;                    ─────────────►    SQL cursor is opened

                           ◄─────────────    Query block with 100
                                             rows is returned
FETCH C1 INTO ...;

FETCH C1 INTO ...;



INSERT INTO ...;           ─────────────►    Server processes
                                             INSERT statement
                           ◄─────────────
```
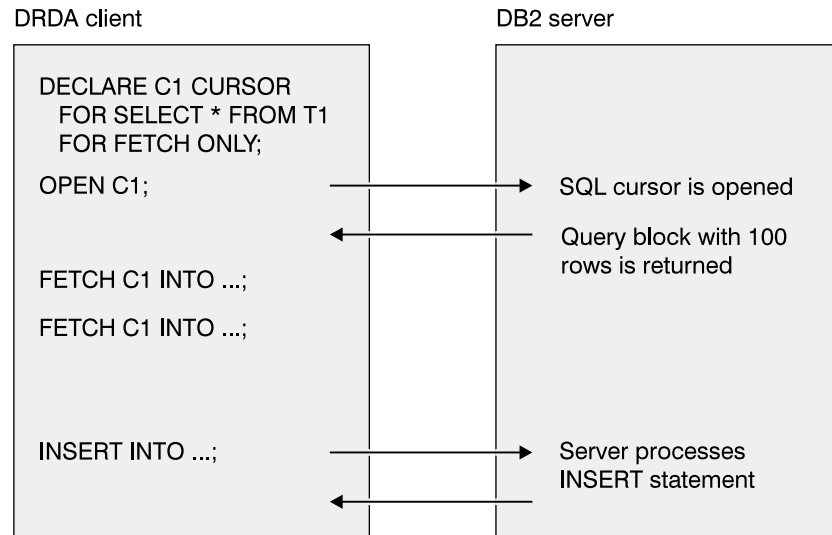
Figure 123. Message flows without OPTIMIZE FOR 1000 ROWS

In Figure 124 on page 388, the DRDA client opens a cursor and fetches rows from the cursor using OPTIMIZE FOR *n* ROWS. Both the DRDA client and the DB2 server are configured to support multiple DRDA query blocks. At some time before the end of the query result set, the application issues an SQL INSERT. Because OPTIMIZE FOR *n* ROWS is being used, the DRDA connection is not available when the SQL INSERT is issued because the connection is still being used to receive the DRDA query blocks for 1000 rows of data. This causes two performance problems:

- Application elapsed time can increase if the DRDA client waits for a large query result set to be transmitted, before the DRDA connection can be used for other SQL statements. Figure 124 on page 388 shows how an SQL INSERT statement can be delayed because of a large query result set.

- If the application closes the cursor before fetching all the rows in the SQL result set, the server might fetch a large number of rows unnecessarily.

```
           DRDA client                          DB2 server

    DECLARE C1 CURSOR
      FOR SELECT * FROM T1
      OPTIMIZE FOR
      1000 ROWS;

    OPEN C1;                  ─────────────────▶  SQL cursor is opened

                             ◀─────────────────   Query block with 100
    FETCH C1 INTO ...;                             rows is returned

    FETCH C1 INTO ...;       ◀─────────────────   Query block with 100
                .                                  rows is returned
                .
                .            ◀─────────────────   Query block with 100
    INSERT INTO ...;                               rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ◀─────────────────   Query block with 100
                                                   rows is returned

                             ─────────────────▶   Server processes
                             ◀─────────────────   INSERT statement
```

Figure 124. Message flows with OPTIMIZE FOR 1000 ROWS

Recommendation: OPTIMIZE FOR *n* ROWS should be used to increase the
number of DRDA query blocks only in applications that have all of these attributes:

- The application fetches a large number of rows from a read-only query.
- The application rarely closes the SQL cursor before fetching the entire query
  result set.
- The application does not issue statements other than FETCH to the DB2 server
  while the SQL cursor is open.
- The application does not execute FETCH statements for multiple cursors that are
  open concurrently and defined with OPTIMIZE FOR *n* ROWS.
- The application does not need to scroll randomly through the data. OPTIMIZE
  FOR *n* ROWS has no effect on a scrollable cursor.

For more information on OPTIMIZE FOR *n* ROWS, see "Minimizing overhead for
retrieving few rows: OPTIMIZE FOR n ROWS" on page 656.

Specifying FETCH FIRST n ROWS ONLY

You can use the FETCH FIRST *n* ROWS ONLY clause of a SELECT statement to
limit the number of rows that are returned to a client program. FETCH FIRST *n*
ROWS ONLY improves performance of DRDA client applications when the client
needs no more than *n* rows from a potentially large result table.

If you specify FETCH FIRST *n* ROWS ONLY in a SELECT statement, OPTIMIZE FOR *n* ROWS is implied. If you specify FETCH FIRST *n* ROWS ONLY and OPTIMIZE FOR *m* ROWS in a SELECT statement, DB2 uses the smaller of *n* and *m* in the OPTIMIZE clause.

Fast implicit close and FETCH FIRST n ROWS ONLY: *Fast implicit close* means that during a distributed query, the DB2 server automatically closes the cursor after it prefetches the *n*th row if you specify FETCH FIRST *n* ROWS ONLY, or when there are no more rows to return. Fast implicit close can improve performance because it saves an additional network transmission between the client and the server.

DB2 uses fast implicit close when the following conditions are true:
- The query uses limited block fetch.
- The query retrieves no LOBs.
- The cursor is not a scrollable cursor.
- Either of the following conditions is true:
  - The cursor is declared WITH HOLD, and the package or plan that contains the cursor is bound with the KEEPDYNAMIC(YES) option.
  - The cursor is not defined WITH HOLD.

When you use FETCH FIRST *n* ROWS ONLY, and DB2 does a fast implicit close, the DB2 server closes the cursor after it prefetches the *n*th row, or when there are no more rows to return.

Example: FETCH FIRST n ROWS ONLY: In a DRDA environment, the following SQL statement causes DB2 to prefetch 16 rows of the result table.

```
SELECT * FROM EMP
 OPTIMIZE FOR 1 ROW ONLY;
```

Suppose that you need only one row of the result table. To avoid 15 unnecessary prefetches, add the FETCH FIRST 1 ROW ONLY clause:

```
SELECT * FROM EMP
 OPTIMIZE FOR 1 ROW ONLY
 FETCH FIRST 1 ROW ONLY;
```

DB2 for OS/390 and z/OS support for the rowset parameter

The rowset parameter can be used in ODBC and JDBC applications on some platforms to limit the number of rows that are returned from a fetch operation. If a DRDA requester sends the rowset parameter to a DB2 for OS/390 and z/OS server, the server does the following things:
- Returns no more than the number of rows in the rowset parameter
- Returns extra query blocks if the value of field EXTRA BLOCKS SRV in the DISTRIBUTED DATA FACILITY PANEL 2 installation panel on the server allows extra query blocks to be returned
- Processes the FETCH FIRST *n* ROWS ONLY clause, if it is specified
- Does not process the OPTIMIZE FOR *n* ROWS clause

Accessing data with a scrollable cursor when the requester is down-level

If a DB2 for OS/390 and z/OS server processes an OPEN cursor statement for a scrollable cursor, and the OPEN cursor statement comes from a requester that

does not support scrollable cursors, the DB2 for OS/390 and z/OS server returns an SQL error. However, if a stored procedure at the server uses a scrollable cursor to return a result set, the down-level requester can access data through that cursor. The DB2 for OS/390 and z/OS server converts the scrollable result set cursor to a non-scrollable cursor. The requester can retrieve the data using sequential FETCH statements.

Maintaining data currency

Cursors used in block fetch operations bound with cursor stability are particularly vulnerable to reading data that has already changed. In a block fetch, database access speeds ahead of the application to prefetch rows. During that time the cursor could close, and the locks be released, before the application receives the data. Thus, it is possible for the application to fetch a row of values that no longer exists, or to miss a recently inserted row. In many cases, that is acceptable; a case for which it is *not* acceptable is said to require *data currency*.

How to prevent block fetching: If your application requires data currency for a cursor, you need to prevent block fetching for the data it points to. To prevent block fetching for a distributed cursor, declare the cursor with the FOR UPDATE or FOR UPDATE OF clause.

Copying a table from a remote location

To copy a table from one location to another, you can either write your own application program or use the DB2 DataPropagator product.

Transmitting mixed data

If you transmit mixed data between your local system and a remote system, put the data in varying-length character strings instead of fixed-length character strings.

When ASCII MIXED data or Unicode MIXED data is converted to EBCDIC MIXED, the converted string is longer than the source string. An error occurs if that conversion is done to a fixed-length input host variable. The remedy is to use a varying-length string variable with a maximum length that is sufficient to contain the expansion.

Identifying the server at run time
The special register CURRENT SERVER contains the location name of the system you are connected to. You can assign that name to a host variable with a statement like this:

```
EXEC SQL SET :CS = CURRENT SERVER;
```

Retrieving data from ASCII or Unicode tables

When you perform a distributed query, the server determines the encoding scheme of the result table. When a distributed query against an ASCII or Unicode table arrives at the DB2 for OS/390 and z/OS server, the server indicates in the reply message that the columns of the result table contain ASCII or Unicode data, rather than EBCDIC data. The reply message also includes the CCSIDs of the data to be returned. The CCSID of data from a column is the CCSID that was in effect when the column was defined.

The encoding scheme in which DB2 returns data depends on two factors:
• The encoding scheme of the requesting system.
  If the requester is ASCII or Unicode, the returned data is ASCII or Unicode. If the requester is EBCDIC, the returned data is EBCDIC, even though it is stored at

the server as ASCII or Unicode. However, if the SELECT statement that is used to retrieve the data contains an ORDER BY clause, the data displays in ASCII or Unicode order.

- Whether the application program overrides the CCSID for the returned data. There are several ways to do this:

  - For static SQL

    You can bind a plan or package with the ENCODING bind option to control the CCSIDs for all static data in that plan or package. For example, if you specify ENCODING(UNICODE) when you bind a package at a remote DB2 for OS/390 and z/OS system, the data that is returned in host variables from the remote system is encoded in the default Unicode CCSID for that system.

    See Chapter 2 of *DB2 Command Reference* for more information on the ENCODING bind options.

  - For static or dynamic SQL

    An application program can specify overriding CCSIDs for individual host variables in DECLARE VARIABLE statements. See "Changing the coded character set ID of host variables" on page 72 for information on how to specify the CCSID for a host variable.

    An application program that uses an SQLDA can specify an overriding CCSID for the returned data in the SQLDA . When the application program executes a FETCH statement, you receive the data in the CCSID that is specified in the SQLDA. See "Changing the CCSID for retrieved data" on page 517 for information on how to specify an overriding CCSID in an SQLDA.

Considerations for moving from DB2 private protocol access to DRDA access

Recommendation: Move from DB2 private protocol access to DRDA access whenever possible. Because DB2 supports three-part names, you can move to DRDA access without modifying your applications. For any application that uses DB2 private protocol access, follow these steps to make the application use DRDA access:

1. Determine which locations the application accesses.

   For static SQL applications, you can do this by searching for all SQL statements that include three-part names and aliases for three-part names. For three-part names, the high-level qualifier is the location name. For potential aliases, query catalog table SYSTABLES to determine whether the object is an alias, and if so, the location name of the table that the alias represents. For example:

   ```
   SELECT NAME, CREATOR, LOCATION, TBCREATOR, TBNAME
     FROM SYSIBM.SYSTABLES
     WHERE NAME='name'
     AND TYPE='A';
   ```

   where *name* is the potential alias.

   For dynamic SQL applications, bind packages at all remote locations that users might access with three-part names.

2. Bind the application into a package at every location that is named in the application. Also bind a package locally.

   For an application that uses explicit CONNECT statements to connect to a second site and then accesses a third site using a three-part name, bind a package at the second site with DBPROTOCOL(DRDA), and bind another package at the third site.

3. Bind all remote packages into a plan with the local package or DBRM. Bind this plan with the option DBPROTOCOL(DRDA).

4. Ensure that aliases resolve correctly.

   For DB2 private protocol access, DB2 resolves aliases at the requester site. For DRDA access, however, DB2 resolves aliases at the site where the package executes. Therefore, you might need to define aliases for three-part names at remote locations.

   For example, suppose you use DRDA access to run a program that contains this statement:

   ```
   SELECT * FROM MYALIAS;
   ```

   MYALIAS is an alias for LOC2.MYID.MYTABLE. DB2 resolves MYALIAS at the local site to determine that this statement needs to run at LOC2 but does not send the resolved name to LOC2. When the statement executes at LOC2, DB2 resolves MYALIAS using the catalog at LOC2. If the catalog does not contain the alias MYID.MYTABLE for MYALIAS, the SELECT statement does not execute successfully.

   This situation can become more complicated if you use three-part names to access DB2 objects from remote sites. For example, suppose you are connected explicitly to LOC2, and you use DRDA access to execute the following statement:

   ```
   SELECT * FROM YRALIAS;
   ```

   YRALIAS is an alias for LOC3.MYID.MYTABLE. When this SELECT statement executes at LOC3, both LOC2 and LOC3 must have an alias YRALIAS that resolves to MYID.MYTABLE at location LOC3.

5. If you use the resource limit facility at the remote locations that are specified in three-part names to control the amount of time that distributed dynamic SQL statements run, you must modify the resource limit specification tables at those locations.

   For DB2 private protocol access, you specify plan names to govern SQL statements that originate at a remote location. For DRDA access, you specify package names for this purpose. Therefore, you must add rows to your resource limit specification tables at the remote locations for the packages you bound for DRDA access with three-part names. You should also delete the rows that specify plan names for DB2 private protocol access.

   For more information on the resource limit facility, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Part 5. Developing your application

Chapter 20. Preparing an application program to run

There are several types of DB2 applications, each of which require different methods of program preparation:
- Applications that contain embedded static or dynamic SQL statements
- Applications that contain ODBC calls
- Applications in interpreted languages, such as REXX
- Java applications, which can contain JDBC calls or embedded SQL statements

Before you can run DB2 applications of the first type, you must precompile, compile, link-edit, and bind them.

Productivity hint: To avoid rework, first test your SQL statements using SPUFI, then compile your program *without* SQL statements and resolve all compiler errors. Then proceed with the preparation and the DB2 precompile and bind steps.

Because most compilers do not recognize SQL statements, you must use the DB2 precompiler before you compile the program to prevent compiler errors. The precompiler scans the program and returns a modified source code, which you can then compile and link edit. The precompiler also produces a DBRM (database request module). Bind this DBRM to a package or plan using the BIND subcommand. (For information on packages and plans, see "Chapter 16. Planning for DB2 program preparation" on page 313.) When you complete these steps, you can run your DB2 application.

This chapter details the steps to prepare your application program to run. It includes instructions for the main steps for producing an application program, additional steps you might need, and steps for rebinding.

Steps in program preparation

The following sections provide details on preparing and running a DB2 application:
"Step 1: Process SQL statements" on page 396
"Step 2: Compile (or assemble) and link-edit the application" on page 408
"Step 3: Bind the application" on page 409
"Step 4: Run the application" on page 421.

As described in "Chapter 16. Planning for DB2 program preparation" on page 313, binding a package is not necessary in all cases. In these instructions, it is assumed that you bind some of your DBRMs into packages and include a package list in your plan.

If you use CICS, you might need additional steps; see:
- "Translating Command-Level Statements" on page 407
- "Define the program to CICS and to the RCT" on page 408
- "Make a New Copy of the Program" on page 425

For information on running REXX programs, which you do not prepare for execution, see "Running a DB2 REXX application" on page 425.

For information on preparing and executing Java programs, see *DB2 Application Programming Guide and Reference for Java*.

There are several ways to control the steps in program preparation. They are described in "Using JCL procedures to prepare applications" on page 425.

Step 1: Process SQL statements

One step in preparing an SQL application to run is processing SQL statements in the program. The SQL statements must be replaced with calls to DB2 language interface modules, and a DBRM must be created. For assembler, C, FORTRAN, or PL/I applications, use the DB2 precompiler to prepare the SQL statements. For COBOL, you can use one of the following techniques to process SQL statements:

- Use the DB2 precompiler before you compile your program.

  You can use this technique with any version of COBOL.

- Use the SQL statement coprocessor as you compile your program.

  You invoke the SQL statement coprocessor by specifying the SQL compiler option.You need IBM COBOL for OS/390 & VM Version 2 Release 2 or later to use this technique. For more information on using the COBOL SQL statement coprocessor, see IBM COBOL for OS/390 & VM Programming Guide.

In this section, references to an *SQL statement processor* apply to either the DB2 precompiler or an SQL statement coprocessor. References to *the DB2 precompiler* apply specifically to the precompiler that is provided with DB2.

CICS

If the application contains CICS commands, you must translate the program before you compile it. (See "Translating command-level statements in a CICS program" on page 407.)

Using the DB2 precompiler

To start the precompile process, use one of the following methods:

- DB2I panels. Use the Precompile panel or the DB2 Program Preparation panels.
- The DSNH command procedure (a TSO CLIST). For a description of that CLIST, see Chapter 2 of *DB2 Command Reference*.
- JCL procedures supplied with DB2. See "Available JCL procedures" on page 425 for more information on this method.

When you precompile your program, DB2 does not need to be active. The precompiler does not validate the names of tables and columns that are used in SQL statements. However, the precompiler checks table and column references against SQL DECLARE TABLE statements in the program. Therefore, you should use DCLGEN to obtain accurate SQL DECLARE TABLE statements.

You might need to precompile and compile program source statements several times before they are error-free and ready to link-edit. During that time, you can get complete diagnostic output from the DB2 precompiler by specifying the SOURCE and XREF precompiler options.

Input to the precompiler: The primary input for the precompiler consists of statements in the host programming language and embedded SQL statements.

> **Important**
>
> The size of a source program that DB2 can precompile is limited by the region size and the virtual memory available to the precompiler. The maximum region size and memory available to the DB2 precompiler is usually around 8 MB, but these amounts vary with each system installation.

You can use the SQL INCLUDE statement to get secondary input from the include library, SYSLIB. The SQL INCLUDE statement reads input from the specified member of SYSLIB until it reaches the end of the member.

Another preprocessor, such as the PL/I macro preprocessor, can generate source statements for the precompiler. Any preprocessor that runs before the precompiler must be able to pass on SQL statements. Similarly, other preprocessors can process the source code, after you precompile and before you compile or assemble.

There are limits on the forms of source statements that can pass through the precompiler. For example, constants, comments, and other source syntax that are not accepted by the host compilers (such as a missing right brace in C) can interfere with precompiler source scanning and cause errors. You might want to run the host compiler before the precompiler to find the source statements that are unacceptable to the host compiler. At this point you can ignore the compiler error messages for SQL statements. After the source statements are free of unacceptable compiler errors, you can then perform the normal DB2 program preparation process for that host language.

The following restrictions apply to the only to the DB2 precompiler:
- You must write host language statements and SQL statements using the same margins, as specified in the precompiler option MARGINS.
- The input data set, SYSIN, must have the attributes RECFM F or FB, LRECL 80.
- SYSLIB must be a partitioned data set, with attributes RECFM F or FB, LRECL 80.
- Input from the INCLUDE library cannot contain other precompiler INCLUDE statements.

Output from the precompiler: The following sections describe various kinds of output from the precompiler.

Listing output: The output data set, SYSPRINT, used to print output from the precompiler, has an LRECL of 133 and a RECFM of FBA. Statement numbers in the output of the precompiler listing always display as they appear in the listing. However, DB2 stores statement numbers greater than 32767 as 0 in the DBRM.

The DB2 precompiler writes the following information in the SYSPRINT data set:
- Precompiler source listing

  If the DB2 precompiler option SOURCE is specified, a source listing is produced. The source listing includes precompiler source statements, with line numbers that are assigned by the precompiler.
- Precompiler diagnostics

  The precompiler produces diagnostic messages that include precompiler line numbers of statements that have errors.

- Precompiler cross-reference listing

  If the DB2 precompiler option XREF is specified, a cross-reference listing is produced. The cross-reference listing shows the precompiler line numbers of SQL statements that refer to host names and columns.

Terminal diagnostics: If a terminal output file, SYSTERM, is present, the DB2 precompiler writes diagnostic messages to it. A portion of the source statement accompanies the messages in this file. You can often use the SYSTERM file instead of the SYSPRINT file to find errors.

Modified source statements: The DB2 precompiler writes the source statements that it processes to SYSCIN, the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, and LRECL 80. The modified source code contains calls to the DB2 language interface. The SQL statements that the calls replace appear as comments.

Database request modules: The major output from the precompiler is a database request module (DBRM). That data set contains the SQL statements and host variable information extracted from the source program, along with information that identifies the program and ties the DBRM to the translated source statements. It becomes the input to the bind process.

The data set requires space to hold all the SQL statements plus space for each host variable name and some header information. The header information alone requires approximately two records for each DBRM, 20 bytes for each SQL record, and 6 bytes for each host variable. For an exact format of the DBRM, see the DBRM mapping macro, DSNXDBRM in library *prefix*.SDSNMACS. The DCB attributes of the data set are RECFM FB, LRECL 80. The precompiler sets the characteristics. You can use IEBCOPY, IEHPROGM, TSO commands COPY and DELETE, or other PDS management tools for maintaining these data sets.

The DB2 language preparation procedures in job DSNTIJMV use the DISP=OLD parameter to enforce data integrity. However, when the installation process converts the DISP=OLD parameter for the DBRM library data set to DISP=SHR, which can cause data integrity problems when you run multiple precompilation jobs. If you plan to run multiple precompilation jobs and are not using DFSMSdfp's partitioned data set extended (PDSE), you must change the DB2 language preparation procedures (DSNHCOB, DSNHCOB2, DSNHICOB, DSNHFOR, DSNHC, DSNHPLI, DSNHASM, DSNHSQL) to specify the DISP=OLD parameter instead of the DISP=SHR parameter.

Binding on another system: It is not necessary to precompile the program on the same DB2 system on which you bind the DBRM and run the program. In particular, you can bind a DBRM at the current release level and run it on a DB2 subsystem at the previous release level, if the original program does not use any properties of DB2 that are unique to the current release. Of course, you can run applications on the current release that were previously bound on systems at the previous release level.

Using the COBOL SQL statement coprocessor

The COBOL SQL statement coprocessor performs DB2 precompiler functions at compile time. In addition, the SQL statement coprocessor lifts some of the DB2 precompiler's restrictions on SQL programs.When you process SQL statements with the COBOL SQL statement coprocessor, you can do the following things in your program:

- Use fully-qualified names for structured host variables

- Include SQL statements at any level of a nested COBOL program, instead of in only the top-level source file
- Use nested SQL INCLUDE statements
- Use COBOL REPLACE statements to replace text strings in SQL statements

To use the SQL statement coprocessor, you need to do the following things:
- Specify the following options when you compile your program:
  - SQL

    The `SQL` compiler option indicates that you want the compiler to invoke the SQL statement coprocessor. Specify a list of SQL processing options in parentheses after the SQL keyword. Table 48 on page 400 lists the options that you can specify.

    For example, suppose that you want to process SQL statements as you compile a COBOL program. In your program, the apostrophe is the string delimiter in SQL statements, and the SQL statements conform to DB2 rules. This means that you need to specify the APOSTSQL and STDSQL(NO) options. Therefore, you need to include this option in your compile step:

    `SQL("APOSTSQL STDSQL(NO)")`
  - LIB

    You need to specify the LIB option when you specify the SQL option, whether or not you have any COPY, BASIS, or REPLACE statements in your program.
  - SIZE(*nnnnnn*)

    You might need to increase the SIZE value so that the user region is large enough for the SQL statement coprocessor. Do not specify SIZE(MAX).
- Include DD statements for the following data sets in the JCL for your compile step:
  - DB2 load library (*prefix*.SDSNLOAD)

    The SQL statement coprocessor calls DB2 modules to do the SQL statement processing. You therefore need to include the name of the DB2 load library data set in the STEPLIB concatenation for the compile step.
  - DBRM library

    The SQL statement coprocessor produces a DBRM. DBRMs and the DBRM library are described in "Output from the precompiler" on page 397. You need to include a DBRMLIB DD statement that specifies the DBRM library data set.
  - Library for SQL INCLUDE statements

    If your program contains SQL INCLUDE *member-name* statements that specify secondary input to the source program, you need to include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compile step.

Options for SQL statement processing

To control the DB2 precompiler or an SQL statement coprocessor, you specify options when you use it. The options specify how the SQL statement processor interprets or processes its input, and how it presents its output.

If you use the DB2 precompiler, you can specify SQL processing options in one of the following ways:
- With DSNH operands
- With the PARM.PC option of the EXEC JCL statement
- In DB2I panels

If you use the COBOL SQL statement coprocessor, you specify the coprocessor options as the argument of the SQL compiler option.

DB2 assigns default values for any SQL processing options for which you do not explicitly specify a value. Those defaults are the values that are specified in the APPLICATION PROGRAMMING DEFAULTS installation panels.

Table of SQL processing options: Table 48 shows the options you can specify when you use the DB2 precompiler or an SQL statement coprocessor. The table also includes abbreviations for those options.

The table uses a vertical bar (|) to separate mutually exclusive options, and brackets ([ ]) to indicate that you can sometimes omit the enclosed option.

Table 48. SQL processing options

Option Keyword	Meaning
APOST[2]	Recognizes the apostrophe (') as the string delimiter *within host language statements*. The option is not available in all languages; see Table 50 on page 406.
	APOST and QUOTE are mutually exclusive options. The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If STRING DELIMITER is the apostrophe ('), APOST is the default.
APOSTSQL	Recognizes the apostrophe (') as the string delimiter and the quotation mark (") as the SQL escape character *within SQL statements*. If you have a COBOL program and you specify SQLFLAG, then you should also specify APOSTSQL.
	APOSTSQL and QUOTESQL are mutually exclusive options. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If SQL STRING DELIMITER is the apostrophe ('), APOSTSQL is the default.
ATTACH(TSO|CAF| RRSAF)	Specifies the attachment facility that the application uses to access DB2. TSO, CAF, and RRSAF applications that load the attachment facility can use this option to specify the correct attachment facility, instead of coding a dummy DSNHLI entry point.
	This option is not available for FORTRAN applications.
	The default is ATTACH(TSO).
COMMA	Recognizes the comma (,) as the decimal point indicator in decimal or floating point literals in the following cases:
	• For static SQL statements in COBOL programs
	• For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior.
	COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 when DB2 is installed.
CONNECT(2|1) CT(2|1)	Determines whether to apply type 1 or type 2 CONNECT statement rules. CONNECT(2) Default: Apply rules for the CONNECT (Type 2) statement. CONNECT(1) Apply rules for the CONNECT (Type 1) statement
	If you do not specify the CONNECT option when you precompile a program, the rules of the CONNECT (Type 2) statement apply. See "Precompiler options" on page 374 for more information about this option, and Chapter 5 of *DB2 SQL Reference* for a comparison of CONNECT (Type 1) and CONNECT (Type 2).

Table 48. SQL processing options (continued)

Option Keyword	Meaning
DATE(ISO|USA|EUR JIS|LOCAL)	Specifies that date output should always return in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 2 of *DB2 SQL Reference*. The default is in the field DATE FORMAT on Application Programming Defaults Panel 2 when DB2 is installed. You cannot use the LOCAL option unless you have a date exit routine.
DEC(15|31)	Specifies the maximum precision for decimal arithmetic operations. See "Using 15-digit and 31-digit precision for decimal numbers" on page 13. The default is in the field DECIMAL ARITHMETIC on Application Programming Defaults Panel 1 when DB2 is installed.
FLAG(I|W|E|S)	Suppresses diagnostic messages below the specified severity level (Informational, Warning, Error, and Severe error for severity codes 0, 4, 8, and 12 respectively). The default setting is FLAG(I).
FLOAT(S390|IEEE)	Determines whether the contents of floating point host variables in C, C++, or assembler programs are in IEEE floating point format or System/390 floating point format. DB2 ignores this option if the value of HOST is anything other than ASM, C, or CPP. The default setting is FLOAT(S390).
GRAPHIC	Indicates that the source code might use mixed data, and that X'0E' and X'0F' are special control characters (shift-out and shift-in) for EBCDIC data. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on Application Programming Defaults Panel 1 when DB2 is installed.
HOST[2](ASM|C[(FOLD)]| CPP[(FOLD)]| COBOL|COB2|IBMCOB| PLI|FORTRAN)	Defines the host language containing the SQL statements. Use COBOL for OS/VS COBOL only. Use COB2 for VS COBOL II. Use IBMCOB for IBM SAA AD/Cycle COBOL/370 and IBM COBOL for MVS & VM. For C, specify: • C if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • C(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase For C++, specify: • CPP if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • CPP(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase If you omit the HOST option, the DB2 precompiler issues a level-4 diagnostic message and uses the default value for this option. The default is in the field LANGUAGE DEFAULT on Application Programming Defaults Panel 1 when DB2 is installed. This option also sets the language-dependent defaults; see Table 50 on page 406.

Table 48. SQL processing options (continued)

Option Keyword	Meaning
LEVEL[(*aaaa*)] L	Defines the level of a module, where *aaaa* is any alphanumeric value of up to seven characters. This option is not recommended for general use, and the DSNH CLIST and the DB2I panels do not support it. For more information, see "Setting the program level" on page 415. For assembler, C, C++, FORTRAN, and PL/I, you can omit the suboption *(aaaa)*. The resulting consistency token is blank. For COBOL, you need to specify the suboption.
LINECOUNT[1,2](*n*) LC	Defines the number of lines per page to be *n* for the DB2 precompiler listing. This includes header lines inserted by the DB2 precompiler. The default setting is LINECOUNT(60).
MARGINS[2](m,n[,c]) MAR	Specifies what part of each source record contains host language or SQL statements; and, for assembler, where column continuations begin. The first option (m) is the beginning column for statements. The second option (n) is the ending column for statements. The third option (c) specifies for assembler where continuations begin. Otherwise, the DB2 precompiler places a continuation indicator in the column immediately following the ending column. Margin values can range from 1 to 80. Default values depend on the HOST option you specify; see Table 50 on page 406. The DSNH CLIST and the DB2I panels do not support this option. In assembler, the margin option must agree with the ICTL instruction, if presented in the source.
NOFOR	In static SQL, eliminates the need for the FOR UPDATE or FOR UPDATE OF clause in DECLARE CURSOR statements. When you use NOFOR, your program can make positioned updates to any columns that the program has DB2 authority to update. When you do not use NOFOR, if you want to make positioned updates to any columns that the program has DB2 authority to update, you need to specify FOR UPDATE with no column list in your DECLARE CURSOR statements. The FOR UPDATE clause with no column list applies to static or dynamic SQL statements. Whether you use or do not use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns named in the clause and specify the acquisition of update locks. You imply NOFOR when you use the option STDSQL(YES). If the resulting DBRM is very large, you might need extra storage when you specify NOFOR or use the FOR UPDATE clause with no column list.
NOGRAPHIC	Indicates the use of X'0E' and X'0F' in a string, but not as control characters. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on Application Programming Defaults Panel 1 when DB2 is installed.
NOOPTIONS[2] NOOPTN	Suppresses the DB2 precompiler options listing.
NOSOURCE[2] NOS	Suppresses the DB2 precompiler source listing. This is the default.
NOXREF[2] NOX	Suppresses the DB2 precompiler cross-reference listing. This is the default.
ONEPASS[2] ON	Processes in one pass, to avoid the additional processing time for making two passes. Declarations must appear before SQL references. Default values depend on the HOST option specified; see Table 50 on page 406. ONEPASS and TWOPASS are mutually exclusive options.

Table 48. SQL processing options (continued)

Option Keyword	Meaning
OPTIONS[2] OPTN	Lists DB2 precompiler options. This is the default.
PERIOD	Recognizes the period (.) as the decimal point indicator in decimal or floating point literals in the following cases: • For static SQL statements in COBOL programs • For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 when DB2 is installed.
QUOTE[2] Q	Recognizes the quotation mark (") as the string delimiter *within host language statements*. This option applies only to COBOL. The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If STRING DELIMITER is the quote (") or DEFAULT, then QUOTE is the default. APOST and QUOTE are mutually exclusive options.
QUOTESQL	Recognizes the quotation mark (") as the string delimiter and the apostrophe (') as the SQL escape character *within SQL statements*. This option applies only to COBOL. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If SQL STRING DELIMITER is the quote (") or DEFAULT, QUOTESQL is the default. APOSTSQL and QUOTESQL are mutually exclusive options.
SOURCE[2] S	Lists DB2 precompiler source and diagnostics.
SQL(ALL|DB2)	Indicates whether the source contains SQL statements other than those recognized by DB2 for OS/390 and z/OS. SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other that DB2 for OS/390 and z/OS using DRDA access. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 for OS/390 and z/OS. Accordingly, the SQL statement processor then accepts statements that do not conform to the DB2 syntax rules. The SQL statement processor interprets and processes SQL statements according to distributed relational database architecture (DRDA) rules. The SQL statement processor also issues an informational message if the program attempts to use IBM SQL reserved words as ordinary identifiers. SQL(ALL) does not affect the limits of the SQL statement processor. SQL(DB2), the default, means to interpret SQL statements and check syntax for use by DB2 for OS/390 and z/OS. SQL(DB2) is recommended when the database server is DB2 for OS/390 and z/OS.

Table 48. SQL processing options (continued)

Option Keyword	Meaning
SQLFLAG(IBM|STD [(*ssname* [,*qualifier*])])	Specifies the standard used to check the syntax of SQL statements. When statements deviate from the standard, the SQL statement processor writes informational messages (flags) to the output listing. The SQLFLAG option is independent of other SQL statement processor options, including SQL and STDSQL. However, if you have a COBOL program and you specify SQLFLAG, then you should also specify APOSTSQL.
	IBM checks SQL statements against the syntax of IBM SQL Version 1. You can also use SAA for this option, as in releases before Version 7.
	STD checks SQL statements against the syntax of the entry level of the ANSI/ISO SQL standard of 1992. You can also use 86 for this option, as in releases before Version 7.
	ssname requests semantics checking, using the specified DB2 subsystem name for catalog access. If you do not specify *ssname*, the SQL statement processor checks only the syntax.
	qualifier specifies the qualifier used for flagging. If you specify a *qualifier*, you must always specify the *ssname* first. If *qualifier* is *not* specified, the default is the authorization ID of the process that started the SQL statement processor.
STDSQL(NO|YES)	Indicates to which rules the output statements should conform.
	STDSQL(YES)[1] indicates that the precompiled SQL statements in the source program conform to certain rules of the SQL standard. STDSQL(NO) indicates conformance to DB2 rules.
	The default is in the field STD SQL LANGUAGE on Application Programming Defaults Panel 2 when DB2 is installed.
	STDSQL(YES) automatically implies the NOFOR option.
TIME(ISO|USA |EUR|JIS|LOCAL)	Specifies that time output always return in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 2 of *DB2 SQL Reference*.
	The default is in the field TIME FORMAT on Application Programming Defaults Panel 2 when DB2 is installed.
	You cannot use the LOCAL option unless you have a time exit routine.
TWOPASS[2] TW	Processes in two passes, so that declarations need not precede references. Default values depend on the HOST option specified; see Table 50 on page 406.
	ONEPASS and TWOPASS are mutually exclusive options.
VERSION(*aaaa*|AUTO)	Defines the version identifier of a package, program, and the resulting DBRM. When you specify VERSION, the SQL statement processor creates a version identifier in the program and DBRM. This affects the size of the load module and DBRM. DB2 uses the version identifier when you bind the DBRM to a plan or package.
	If you do not specify a version at precompile time, then an empty string is the default version identifier. If you specify AUTO, the SQL statement processor uses the consistency token to generate the version identifier. If the consistency token is a timestamp, the timestamp is converted into ISO character format and used as the version identifier. The timestamp used is based on the System/370 Store Clock value. For information on using VERSION, see "Identifying a package version" on page 414.
XREF[2]	Includes a sorted cross-reference listing of symbols used in SQL statements in the listing output.

Table 48. SQL processing options (continued)

Option Keyword	Meaning

Notes:

1. You can use STDSQL(86) as in prior releases of DB2. The SQL statement processor treats it the same as STDSQL(YES).
2. This option is ignored when the COBOL compiler precompiles the application.

Defaults for options of the SQL statement processor: Some SQL statement processor options have defaults based on values specified on the Application Programming Defaults panels. Table 49 shows those options and defaults:

Table 49. IBM-supplied installation default SQL statement processing options. The installer can change these defaults.

Install option (DSNTIPF)	Install default	Equivalent SQL statement processing option	Available SQL statement processing options
STRING DELIMITER	quotation mark (")	QUOTE	APOST QUOTE
SQL STRING DELIMITER	quotation mark (")	QUOTESQL	APOSTSQL QUOTESQL
DECIMAL POINT IS	PERIOD	PERIOD	COMMA PERIOD
DATE FORMAT	ISO	DATE(ISO)	DATE(ISO|USA| EUR|JIS|LOCAL)
DECIMAL ARITHMETIC	DEC15	DEC(15)	DEC(15|31)
MIXED DATA	NO	NOGRAPHIC	GRAPHIC NOGRAPHIC
LANGUAGE DEFAULT	COBOL	HOST(COBOL)	HOST(ASM|C[(FOLD)]| CPP[(FOLD)]| COBOL|COB2|IBMCOB| FORTRAN|PLI)
STD SQL LANGUAGE	NO	STDSQL(NO)	STDSQL(YES|NO|86)
TIME FORMAT	ISO	TIME(ISO)	TIME(IS|USA|EUR| JIS|LOCAL)

Note:

For dynamic SQL statements, another application programming default, USE FOR DYNAMICRULES, determines whether DB2 uses the application programming default or the SQL statement processor option for the following install options:
- STRING DELIMITER
- SQL STRING DELIMITER
- DECIMAL POINT IS
- DECIMAL ARITHMETIC
- MIXED DATA

If the value of USE FOR DYNAMICRULES is YES, then dynamic SQL statements use the application programming defaults. If the value of USE FOR DYNAMICRULES is NO, then dynamic SQL statements in packages or plans with bind, define, and invoke behavior use the SQL statement processor options. See "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415 for an explanation of bind, define, and invoke behavior.

Some SQL statement processor options have default values based on the host language. Some options do not apply to some languages. Table 50 on page 406 show the language-dependent options and defaults.

Table 50. Language-dependent DB2 precompiler options and defaults

HOST value	Defaults
ASM	APOST[1], APOSTSQL[1], PERIOD[1], TWOPASS, MARGINS(1,71,16)
C or CPP	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS, MARGINS(1,72)
COBOL, COB2, or IBMCOB	QUOTE[2], QUOTESQL[2], PERIOD, ONEPASS[1], MARGINS(8,72)[1]
FORTRAN	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS[1], MARGINS(1,72)[1]
PLI	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS, MARGINS(2,72)

Note:

1. Forced for this language; no alternative allowed.

2. The default is chosen on Application Programming Defaults Panel 1 when DB2 is installed. The IBM-supplied installation defaults for string delimiters are QUOTE (host language delimiter) and QUOTESQL (SQL escape character). The installer can replace the IBM-supplied defaults with other defaults. The precompiler options you specify override any defaults in effect.

SQL statement processing defaults for dynamic statements: Generally, dynamic statements use the defaults specified on installation panel DSNTIPF. However, if the value of DSNHDECP parameter DYNRULS is NO, then you can use these options for dynamic SQL statements in packages or plans with bind, define, or invoke behavior:

- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- GRAPHIC or NOGRAPHIC
- DEC(15) or DEC(31)

Translating command-level statements in a CICS program

CICS

Translating command-level statements: You can translate CICS applications with the CICS command language translator as a part of the program preparation process. (CICS command language translators are available only for assembler, C, COBOL, and PL/I languages; there is currently no translator for FORTRAN.) Prepare your CICS program in either of these sequences:

Use the DB2 precompiler first, followed by the CICS Command Language Translator. This sequence is the preferred method of program preparation and the one that the DB2I Program Preparation panels support. If you use the DB2I panels for program preparation, you can specify translator options automatically, rather than having to provide a separate option string. For further description of DB2I and its uses in program preparation, see "Using ISPF and DB2 Interactive (DB2I)" on page 431.

Use the CICS command language translator first, followed by the DB2 precompiler. This sequence results in a warning message from the CICS translator for each EXEC SQL statement it encounters. The warning messages have no effect on the result. If you are using double-byte character sets (DBCS), we recommend that you precompile before translating, as described above.

Program and process requirements:

Use the precompiler option NOGRAPHIC to prevent the precompiler from mistaking CICS translator output for graphic data.

If your source program is in COBOL, you must specify a string delimiter that is the same for the DB2 precompiler, COBOL compiler, and CICS translator. The defaults for the DB2 precompiler and COBOL compiler are not compatible with the default for the CICS translator.

If the SQL statements in your source program refer to host variables that a pointer stored in the CICS TWA addresses, you must make the host variables addressable to the TWA before you execute those statements. For example, a COBOL application can issue the following statement to establish addressability to the TWA:

```
    EXEC CICS ADDRESS
      TWA (address-of-twa-area)
    END-EXEC
```

Step 2: Compile (or assemble) and link-edit the application

If you use the DB2 precompiler, your next step in the program preparation process is to compile and link-edit your program. As with the precompile step, you have a choice of methods:
- DB2I panels
- The DSNH command procedure (a TSO CLIST)
- JCL procedures supplied with DB2.
- JCL procedures supplied with a host language compiler.

If you use an SQL statement coprocessor, you process SQL statements as you compile your program. You must use JCL procedures when you use the SQL statement coprocessor.

The purpose of the link edit step is to produce an executable load module. To enable your application to interface with the DB2 subsystem, you must use a link-edit procedure that builds a load module that satisfies these requirements:

> **IMS**
>
> Include the DB2 IMS (Version 1 Release 3 or later) language interface module (DFSLI000). Also, the IMS RESLIB must precede the SDSNLOAD library in the link list, JOBLIB, or STEPLIB concatenations.

> **CICS**
>
> Include the DB2 CICS language interface module (DSNCLI).
>
> You can link DSNCLI with your program in either 24 bit or 31 bit addressing mode (AMODE=31). If your application program runs in 31-bit addressing mode, you should link-edit the DSNCLI stub to your application with the attributes AMODE=31 and RMODE=ANY so that your application can run above the 16M line. For more information on compiling and link-editing CICS application programs, see the appropriate CICS manual.
>
> You also need the CICS EXEC interface module appropriate for the programming language. CICS requires that this module be the first control section (CSECT) in the final load module.

The size of the executable load module that is produced by the link-edit step varies depending on the values that the SQL statement processor inserts into the source code of the program.

For more information on compiling and link-editing, see "Using JCL procedures to prepare applications" on page 425.

For more information on link-editing attributes, see the appropriate MVS manuals. For details on DSNH, see Chapter 2 of *DB2 Command Reference*.

Step 3: Bind the application

You must bind the DBRM produced by the SQL statement processor to a plan or package before your DB2 application can run. A plan can contain DBRMs, a package list specifying packages or collections of packages, or a combination of DBRMs and a package list. The plan must contain at least one package or at least one directly-bound DBRM. Each package you bind can contain only one DBRM.

> **Exception**
>
> You do not need to bind a DBRM if the only SQL statement in the program is SET CURRENT PACKAGESET.

Because you do not need a plan or package to execute the SET CURRENT PACKAGESET statement, the ENCODING bind option does not affect the SET CURRENT PACKAGESET statement. An application that needs to provide a host variable value in an encoding scheme other than the system default encoding scheme must use the DECLARE VARIABLE statement to specify the encoding scheme of the host variable.

You must bind plans locally, whether or not they reference packages that run remotely. However, you must bind the packages that run at remote locations at those remote locations.

From a DB2 requester, you can run a plan by naming it in the RUN subcommand, but you cannot run a package directly. You must include the package in a plan and then run the plan.

Binding a DBRM to a package

When you bind a package, you specify the collection to which the package belongs. The collection is not a physical entity, and you do not create it; the collection name is merely a convenient way of referring to a group of packages.

To bind a package, you must have the proper authorization.

Binding packages at a remote location: When your application accesses data through DRDA access, you must bind packages on the systems on which they will run. At your local system you must bind a plan whose package list includes all those packages, local and remote.

To bind a package at a remote DB2 system, you must have all the privileges or authority there that you would need to bind the package on your local system. To bind a package at another type of a system, such as SQL/DS, you need any privileges that system requires to execute its SQL statements and use its data objects.

The bind process for a remote package is the same as for a local package, except that the local communications database must be able to recognize the location name you use as resolving to a remote location. To bind the DBRM PROGA at the location PARIS, in the collection GROUP1, use:

```
BIND PACKAGE(PARIS.GROUP1)
    MEMBER(PROGA)
```

Then, include the remote package in the package list of a local plan, say PLANB, by using:

```
BIND PLAN (PLANB)
    PKLIST(PARIS.GROUP1.PROGA)
```

The ENCODING bind option has the following effect on a remote application:

- If you bind a package locally, which is recommended, and you specify the ENCODING bind option for the local package, the ENCODING bind option for the local package applies to the remote application.
- If you do not bind a package locally, and you specify the ENCODING bind option for the plan, the ENCODING bind option for the plan applies to the remote application.
- If you do not specify an ENCODING bind option for the package or plan at the local site, the value of APPLICATION ENCODING that was specified on installation panel DSNTIPF at the local site applies to the remote application.

When you bind or rebind, DB2 checks authorizations, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 does not read or update catalogs or check authorizations at the local site.

If you specify the option EXPLAIN(YES) and you do not specify the option SQLERROR(CONTINUE), then PLAN_TABLE must exist at the location specified on the BIND or REBIND subcommand. This location could also be the default location.

If you bind with the option COPY, the COPY privilege must exist locally. DB2 performs authorization checking, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 reads the catalog records related to the copied package at the local site. If the local site is installed with time or date format LOCAL, and a package is created at a remote site using the COPY option, the COPY option causes DB2 at the remote site to convert values returned from the remote site in ISO format, unless an SQL statement specifies a different format.

Once you bind a package, you can rebind, free, or bind it with the REPLACE option using either a local or a remote bind.

Turning an existing plan into packages to run remotely: If you have used DB2 before, you might have an existing application that you want to run at a remote location, using DRDA access. To do that, you need to rebind the DBRMs in the current plan as packages at the remote location. You also need a new plan that includes those remote packages in its package list.

Follow these instructions for each remote location:
1. Choose a name for a collection to contain all the packages in the plan, say REMOTE1. (You can use more than one collection if you like, but one is enough.)
2. Assuming that the server is a DB2 system, at the remote location execute:
   a. GRANT CREATE IN COLLECTION REMOTE1 TO *authorization-name*;
   b. GRANT BINDADD TO *authorization-name*;

   where *authorization-name* is the owner of the package.
3. Bind *each* DBRM as a package at the remote location, using the instructions under "Binding packages at a remote location" on page 410. Before run time, the package owner must have all the data access privileges needed at the remote location. If the owner does not yet have those privileges when you are binding, use the VALIDATE(RUN) option. The option lets you create the package, even if the authorization checks fail. DB2 checks the privileges again at run time.
4. Bind a new application plan at your local DB2, using these options:
   ```
   PKLIST (location-name.REMOTE1.*)
   CURRENTSERVER (location-name)
   ```

   where *location-name* is the value of LOCATION, in SYSIBM.LOCATIONS at your local DB2, that denotes the remote location at which you intend to run. You do not need to bind any DBRMs directly to that plan: the package list is sufficient.

When you now run the existing application at your local DB2, using the new application plan, these things happen:
- You connect immediately to the remote location named in the CURRENTSERVER option.
- When about to run a package, DB2 searches for it in the collection REMOTE1 at the remote location.
- Any UPDATE, DELETE, or INSERT statements in your application affect tables at the remote location.

- Any results from SELECT statements return to your existing application program, which processes them as though they came from your local DB2.

Binding an application plan

Use the BIND PLAN subcommand to bind DBRMs and package lists to a plan. For BIND PLAN syntax and complete descriptions, see Chapter 2 of *DB2 Command Reference*.

Binding DBRMs directly to a plan: A plan can contain DBRMs bound directly to it. To bind three DBRMs—PROGA, PROGB, and PROGC—directly to plan PLANW, use:

```
BIND PLAN(PLANW)
    MEMBER(PROGA,PROGB,PROGC)
```

You can include as many DBRMs in a plan as you wish. However, if you use a large number of DBRMs in a plan (more than 500, for example), you could have trouble maintaining the plan. To ease maintenance, you can bind each DBRM separately as a package, specifying the same collection for all packages bound, and then bind a plan specifying that collection in the plan's package list. If the design of the application prevents this method, see if your system administrator can increase the size of the EDM pool to be at least 10 times the size of either the largest database descriptor (DBD) or the plan, whichever is greater.

Including packages in a package list: To include packages in the package list of a plan, list them after the PKLIST keyword of BIND PLAN. To include an entire collection of packages in the list, use an asterisk after the collection name. For example,

```
PKLIST(GROUP1.*)
```

To bind DBRMs directly to the plan, and also include packages in the package list, use both MEMBER and PKLIST. The example below includes:
- The DBRMs PROG1 and PROG2
- All the packages in a collection called TEST2
- The packages PROGA and PROGC in the collection GROUP1

```
MEMBER(PROG1,PROG2)
PKLIST(TEST2.*,GROUP1.PROGA,GROUP1.PROGC)
```

You must specify MEMBER, PKLIST, or both options. The plan that results consists of one of the following:
- Programs associated with DBRMs in the MEMBER list only
- Programs associated with packages and collections identified in PKLIST only
- A combination of the specifications on MEMBER and PKLIST

Identifying packages at run time

The DB2 precompiler or SQL statement coprocessor identifies each call to DB2 with a *consistency token*. The same token identifies the DBRM that the SQL statement processor produces and the plan or package to which you bound the DBRM. When you run the program, DB2 uses the consistency token in matching the call to DB2 to the correct DBRM.

(Usually, the consistency token is in an internal DB2 format. You can override that token if you wish: see "Setting the program level" on page 415.)

But you need other identifiers also. The consistency token alone uniquely identifies a DBRM bound directly to a plan, but it does not necessarily identify a unique package. When you bind DBRMs directly to a particular plan, you bind each one

only once. But you can bind the same DBRM to many packages, at different locations and in different collections, and then you can include all those packages in the package list of the same plan. All those packages will have the same consistency token. As you might expect, there are ways to specify a particular location or a particular collection at run time.

Identifying the location: When your program executes an SQL statement, DB2 uses the value in the CURRENT SERVER special register to determine the location of the necessary package or DBRM. If the current server is your local DB2 and it does not have a location name, the value is blank.

You can change the value of CURRENT SERVER by using the SQL CONNECT statement in your program. If you do not use CONNECT, the value of CURRENT SERVER is the location name of your local DB2 (or blank, if your DB2 has no location name).

Identifying the collection: When your program executes an SQL statement, DB2 uses the value in the CURRENT PACKAGESET special register as the collection name for a necessary package. To set or change that value within your program, use the SQL SET CURRENT PACKAGESET statement.

If you do not use SET CURRENT PACKAGESET, the value in the register is blank when your application begins to run and remains blank. In that case, the order in which DB2 searches available collections can be important.

When you call a stored procedure, the special register CURRENT PACKAGESET contains the value that you specified for the COLLID parameter when you defined the stored procedure. When the stored procedure returns control to the calling program, DB2 restores CURRENT PACKAGESET to the value it contained before the call.

The order of search: The order in which you specify packages in a package list can affect run-time performance. Searching for the specific package involves searching the DB2 directory, which can be costly. When you use collection-id.* with PKLIST keyword, you should specify first the collections in which DB2 is most likely to find a package.

For example, if you perform the following bind: `BIND PLAN (PLAN1) PKLIST (COL1.*,` `COL2.*, COL3.*, COL4.*)` and you then execute program PROG1, DB2 does the following:
1. Checks to see if there is a PROG1 program bound as part of the plan
2. Searches for COL1.PROG1.*timestamp*
3. If it does not find COL1.PROG1.*timestamp*, searches for COL2.PROG1.*timestamp*
4. If it does not find COL2.PROG1.*timestamp*, searches for COL3.PROG1.*timestamp*
5. If it does not find COL3.PROG1.*timestamp*, searches for COL4.PROG1.*timestamp*.

If the special register CURRENT PACKAGESET is blank, DB2 searches for a DBRM or a package in one of these sequences:
* *At the local location* (if CURRENT SERVER is blank or names that location explicitly), the order is:
    1. All DBRMs bound directly to the plan.

2. All packages already allocated to the plan while the plan is running.
3. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan. DB2 searches for packages in the order that they appear in the package list.
- *At a remote location*, the order is:
  1. All packages already allocated to the plan at that location while the plan is running.
  2. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. DB2 searches for packages in the order that they appear in the package list.

  If you use the BIND PLAN option DEFER(PREPARE), DB2 does not search all collections in the package list. See "Use bind options that improve performance" on page 380 for more information.

If you set the special register CURRENT PACKAGESET, DB2 skips the check for programs that are part of the plan and uses the value of CURRENT PACKAGESET as the collection. For example, if CURRENT PACKAGESET contains COL5, then DB2 uses COL5.PROG1.*timestamp* for the search.

Explicitly specifying the intended collection with the CURRENT PACKAGESET register can avoid a potentially costly search through the package list when there are many qualifying entries.

If the order of search is not important: In many cases, DB2's order of search is not important to you and does not affect performance. For an application that runs only at your local DB2, you can name every package differently and include them all in the same collection. The package list on your BIND PLAN subcommand can read:

```
PKLIST (collection.*)
```

You can add packages to the collection even after binding the plan. DB2 lets you bind packages having the same package name into the same collection only if their version IDs are different.

If your application uses DRDA access, you must bind some packages at remote locations. Use the same collection name at each location, and identify your package list as:

```
PKLIST (*.collection.*)
```

If you use an asterisk for part of a name in a package list, DB2 checks the authorization for the package to which the name resolves at run time. To avoid the checking at run time in the example above, you can grant EXECUTE authority for the entire collection to the owner of the plan before you bind the plan.

Identifying a package version: Sometimes, however, you want to have more than one package with the same name available to your plan. The VERSION option makes that possible. Using VERSION identifies your program with a specific version of a package. If you bind the plan with PKLIST (COLLECT.*), then you can do this:

Step Number	For Version 1	For Version 2
1	Precompile program 1, using VERSION(1).	Precompile program 2, using VERSION(2).

Step Number	For Version 1	For Version 2
2	Bind the DBRM with the collection name COLLECT and your chosen package name (say, PACKA).	Bind the DBRM with the collection name COLLECT and package name PACKA.
3	Link-edit program 1 into your application.	Link-edit program 2 into your application.
4	Run the application; it uses program 1 and PACKA, VERSION 1.	Run the application; it uses program 2 and PACKA, VERSION 2.

You can do that with many versions of the program, without having to rebind the application plan. Neither do you have to rename the plan or change any RUN subcommands that use it.

Setting the program level: To override DB2's construction of the consistency token, use the LEVEL (*aaaa*) option. DB2 uses the value you choose for *aaaa* to generate the consistency token. Although we do not recommend this method for general use and the DSNH CLIST or the DB2 Program Preparation panels do not support it, it allows you to do the following:
1. Change the source code (but not the SQL statements) in the DB2 precompiler output of a bound program.
2. Compile and link-edit the changed program.
3. Run the application without rebinding a plan or package.

Using BIND and REBIND options for packages and plans
This section discusses a few of the more complex bind and rebind options. For syntax and complete descriptions of all of the bind and rebind options, see Chapter 2 of *DB2 Command Reference*.

Using DYNAMICRULES to specify behavior of dynamic SQL statements: The BIND or REBIND option DYNAMICRULES determines what values apply at run time for the following dynamic SQL attributes:
* The authorization ID that is used to check authorization
* The qualifier that is used for unqualified objects
* The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements
* Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements

In addition to the DYNAMICRULES value, the run-time environment of a package controls how dynamic SQL statements behave at run time. The two possible run-time environments are:
* The package runs as part of a stand-alone program.
* The package runs as a stored procedure or user-defined function package, or runs under a stored procedure or user-defined function.

  A package that runs under a stored procedure or user-defined function is a package whose associated program meets one of the following conditions:
  – The program is called by a stored procedure or user-defined function.
  – The program is in a series of nested calls that start with a stored procedure or user-defined function.

The combination of the DYNAMICRULES value and the run-time environment determine the values for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement *behavior*. The four behaviors are:
- Run behavior
- Bind behavior
- Define behavior
- Invoke behavior

Table 51 shows the combination of DYNAMICRULES value and run-time environment that yield each dynamic SQL behavior. Table 52 shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

Table 51. How DYNAMICRULES and the run-time environment determine dynamic SQL statement behavior

DYNAMICRULES value	Behavior of dynamic SQL statements	
	Stand-alone program environment	**User-defined function or stored procedure environment**
BIND	Bind behavior	Bind behavior
RUN	Run behavior	Run behavior
DEFINEBIND	Bind behavior	Define behavior
DEFINERUN	Run behavior	Define behavior
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

Note to Table 51:

The BIND and RUN values can be specified for packages and plans. The other values can be specified only for packages.

Table 52. Definitions of dynamic SQL statement behaviors

Dynamic SQL attribute	Setting for dynamic SQL attributes			
	Bind behavior	**Run behavior**	**Define behavior**	**Invoke behavior**
Authorization ID	Plan or package owner	Current SQLID	User-defined function or stored procedure owner	Authorization ID of invoker[1]
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	Current SQLID	User-defined function or stored procedure owner	Authorization ID of invoker
CURRENT SQLID[2]	Not applicable	Applies	Not applicable	Not applicable
Source for application programming options	Determined by DSNHDECP parameter DYNRULS[3]	Install panel DSNTIPF	Determined by DSNHDECP parameter DYNRULS[3]	Determined by DSNHDECP parameter DYNRULS[3]
Can execute GRANT, REVOKE, CREATE, ALTER, DROP, RENAME?	No	Yes	No	No

Table 52. Definitions of dynamic SQL statement behaviors (continued)

Dynamic SQL attribute	Setting for dynamic SQL attributes			
	Bind behavior	Run behavior	Define behavior	Invoke behavior

Note:

1. If the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs will also be checked if they are needed for the required authorization. Otherwise, only one ID, the ID of the invoker, is checked for the required authorization.

2. DB2 uses the value of CURRENT SQLID as the authorization ID for dynamic SQL statements only for plans and packages that have run behavior. For the other dynamic SQL behaviors, DB2 uses the authorization ID that is associated with each dynamic SQL behavior, as shown in this table.

   The value to which CURRENT SQLID is initialized is independent of the dynamic SQL behavior. For stand-alone programs, CURRENT SQLID is initialized to the primary authorization ID. See Table 35 on page 275 and Table 60 on page 543 for information on initialization of CURRENT SQLID for user-defined functions and stored procedures.

   You can execute the SET CURRENT SQLID statement to change the value of CURRENT SQLID for packages with any dynamic SQL behavior, but DB2 uses the CURRENT SQLID value only for plans and packages with run behavior.

3. The value of DSNHDECP parameter DYNRULS, which you specify in field USE FOR DYNAMICRULES in installation panel DSNTIPF, determines whether DB2 uses the SQL statement processing options or the application programming defaults for dynamic SQL statements. See "Options for SQL statement processing" on page 399 for more information.

For more information on DYNAMICRULES, see Chapter 2 of *DB2 SQL Reference* and Chapter 2 of *DB2 Command Reference*.

Determining the optimal authorization cache size: When DB2 determines that you have the EXECUTE privilege on a plan, package collection, stored procedure, or user-defined function, DB2 can cache your authorization ID. When you run the plan, package, stored procedure, or user-defined function, DB2 can check your authorization more quickly.

Determining the authorization cache size for plans: The CACHESIZE option (optional) allows you to specify the size of the cache to acquire for the plan. DB2 uses this cache for caching the authorization IDs of those users running a plan. DB2 uses the CACHESIZE value to determine the amount of storage to acquire for the authorization cache. DB2 acquires storage from the EDM storage pool. The default CACHESIZE value is 1024 or the size set at install time.

The size of the cache you specify depends on the number of individual authorization IDs actively using the plan. Required overhead takes 32 bytes, and each authorization ID takes up 8 bytes of storage. The minimum cache size is 256 bytes (enough for 28 entries and overhead information) and the maximum is 4096 bytes (enough for 508 entries and overhead information). You should specify size in multiples of 256 bytes; otherwise, the specified value rounds up to the next highest value that is a multiple of 256.

If you run the plan infrequently, or if authority to run the plan is granted to PUBLIC, you might want to turn off caching for the plan so that DB2 does not use unnecessary storage. To do this, specify a value of 0 for the CACHESIZE option.

Any plan that you run repeatedly is a good candidate for tuning using the CACHESIZE option. Also, if you have a plan that a large number of users run concurrently, you might want to use a larger CACHESIZE.

Determining the authorization cache size for packages: DB2 provides a single package authorization cache for an entire DB2 subsystem. The DB2 installer sets the size of the package authorization cache by entering a size in field PACKAGE AUTH CACHE of DB2 installation panel DSNTIPP. A 32KB authorization cache is large enough to hold authorization information for about 375 package collections.

See *DB2 Installation Guide* for more information on setting the size of the package authorization cache.

Determining the authorization cache size for stored procedures and user-defined functions: DB2 provides a single routine authorization cache for an entire DB2 subsystem. The routine authorization cache stores a list of authorization IDs that have the EXECUTE privilege on user-defined functions or stored procedures. The DB2 installer sets the size of the routine authorization cache by entering a size in field ROUTINE AUTH CACHE of DB2 installation panel DSNTIPP. A 32KB authorization cache is large enough to hold authorization information for about 380 stored procedures or user-defined functions.

See *DB2 Installation Guide* for more information on setting the size of the routine authorization cache.

Specifying the SQL rules: Not only does SQLRULES specify the rules under which a type 2 CONNECT statement executes, but it also sets the initial value of the special register CURRENT RULES when the database server is the local DB2. When the server is not the local DB2, the initial value of CURRENT RULES is DB2. After binding a plan, you can change the value in CURRENT RULES in an application program using the statement SET CURRENT RULES.

CURRENT RULES determines the SQL rules, DB2 or SQL standard, that apply to SQL behavior at run time. For example, the value in CURRENT RULES affects the behavior of defining check constraints using the statement ALTER TABLE on a populated table:

- **If CURRENT RULES has a value of STD** and no existing rows in the table violate the check constraint, DB2 adds the constraint to the table definition. Otherwise, an error occurs and DB2 does not add the check constraint to the table definition.

  If the table contains data and is already in a check pending status, the ALTER TABLE statement fails.

- **If CURRENT RULES has a value of DB2**, DB2 adds the constraint to the table definition, defers the enforcing of the check constraints, and places the table space or partition in check pending status.

You can use the statement SET CURRENT RULES to control the action that the statement ALTER TABLE takes. Assuming that the value of CURRENT RULES is initially STD, the following SQL statements change the SQL rules to DB2, add a check constraint, defer validation of that constraint and place the table in check pending status, and restore the rules to STD.

```
EXEC SQL
  SET CURRENT RULES = 'DB2';
EXEC SQL
  ALTER TABLE DSN8710.EMP
    ADD CONSTRAINT C1 CHECK (BONUS <= 1000.0);
EXEC SQL
  SET CURRENT RULES = 'STD';
```

add a check constraint immediately to a populated table, defer constraint validation, place the table in check pending status, and restore the standard option. See "Using table check constraints" on page 201 for information on check constraints.

You can also use the CURRENT RULES in host variable assignments, for example:

```
SET :XRULE = CURRENT RULES;
```

and as the argument of a search-condition, for example:

```
SELECT * FROM SAMPTBL WHERE COL1 = CURRENT RULES;
```

Using packages with dynamic plan selection

CICS

You can use packages and dynamic plan selection together, but when you dynamically switch plans, the following conditions must exist:

- All special registers, including CURRENT PACKAGESET, must contain their initial values.
- The value in the CURRENT DEGREE special register cannot have changed during the current transaction.

The benefit of using dynamic plan selection and packages together is that you can convert individual programs in an application containing many programs and plans, one at a time, to use a combination of plans and packages. This reduces the number of plans per application, and having fewer plans reduces the effort needed to maintain the dynamic plan exit.

Given the following example programs and DBRMs:

Program Name
    DBRM Name
MAIN MAIN
PROGA
    PLANA
PROGB
    PKGB
PROGC
    PLANC

you could create packages and plans using the following bind statements:

```
BIND PACKAGE(PKGB) MEMBER(PKGB)
BIND PLAN(MAIN) MEMBER(MAIN,PLANA) PKLIST(*.PKGB.*)
BIND PLAN(PLANC) MEMBER(PLANC)
```

The following scenario illustrates thread association for a task that runs program MAIN:

Sequence of SQL Statements
    Events

1. EXEC CICS START TRANSID(MAIN)
    TRANSID(MAIN) executes program MAIN.

2. EXEC SQL SELECT...
    Program MAIN issues an SQL SELECT statement. The default dynamic plan exit selects plan MAIN.

3. EXEC CICS LINK PROGRAM(PROGA)

4. EXEC SQL SELECT...
    DB2 does not call the default dynamic plan exit, because the program does not issue a sync point. The plan is MAIN.

```
┌─ CICS (continued) ────────────────────────────────────────────┐
│                                                                │
│ Sequence of SQL Statements                                     │
│         Events                                                 │
│                                                                │
│ 5. EXEC CICS LINK PROGRAM(PROGB)                               │
│                                                                │
│                                                                │
│ 6. EXEC SQL SELECT...                                          │
│         DB2 does not call the default dynamic plan exit, because the program │
│         does not issue a sync point. The plan is MAIN and the program uses   │
│         package PKGB.                                          │
│                                                                │
│ 7. EXEC CICS SYNCPOINT                                         │
│         DB2 calls the dynamic plan exit when the next SQL statement           │
│         executes.                                             │
│                                                                │
│ 8. EXEC CICS LINK PROGRAM(PROGC)                               │
│                                                                │
│                                                                │
│ 9. EXEC SQL SELECT...                                          │
│         DB2 calls the default dynamic plan exit and selects PLANC.            │
│                                                                │
│ 10. EXEC SQL SET CURRENT SQLID = 'ABC'                         │
│                                                                │
│                                                                │
│ 11. EXEC CICS SYNCPOINT                                        │
│         DB2 does not call the dynamic plan exit when the next SQL statement   │
│         executes, because the previous statement modifies the special        │
│         register CURRENT SQLID.                               │
│                                                                │
│ 12. EXEC CICS RETURN                                          │
│         Control returns to program PROGB.                      │
│                                                                │
│ 13. EXEC SQL SELECT...                                         │
│         SQLCODE -815 occurs because the plan is currently PLANC and the       │
│         program is PROGB.                                      │
│                                                                │
└────────────────────────────────────────────────────────────────┘
```

Step 4: Run the application

After you have completed all the previous steps, you are ready to run your application. At this time, DB2 verifies that the information in the application plan and its associated packages is consistent with the corresponding information in the DB2 system catalog. If any destructive changes, such as DROP or REVOKE, occur (either to the data structures that your application accesses or to the binder's authority to access those data structures), DB2 automatically rebinds packages or the plan as needed.

DSN command processor

The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in JES-initiated batch. It uses the TSO attachment facility to access DB2. The DSN command processor provides an alternative method for running programs that access DB2 in a TSO environment.

You can use the DSN command processor implicitly during program development for functions such as:

• Using the declarations generator (DCLGEN)

- Running the BIND, REBIND, and FREE subcommands on DB2 plans and packages for your program
- Using SPUFI (SQL Processor Using File Input) to test some of the SQL functions in the program

The DSN command processor runs with the TSO terminal monitor program (TMP). Because the TMP runs in either foreground or background, DSN applications run interactively or as batch jobs.

The DSN command processor can provide these services to a program that runs under it:
- Automatic connection to DB2
- Attention key support
- Translation of return codes into error messages

Limitations of the DSN command processor: When using DSN services, your application runs under the control of DSN. Because TSO executes the ATTACH macro to start DSN, and DSN executes the ATTACH macro to start a part of itself, your application gains control two task levels below that of TSO.

Because your program depends on DSN to manage your connection to DB2:
- If DB2 is down, your application cannot begin to run.
- If DB2 terminates, your application also terminates.
- An application can use only one plan.

If these limitations are too severe, consider having your application use the call attachment facility or Recoverable Resource Manager Services attachment facility. For more information on these attachment facilities, see "Chapter 29. Programming for the call attachment facility (CAF)" on page 729 and "Chapter 30. Programming for the Recoverable Resource Manager Services attachment facility (RRSAF)" on page 763.

DSN return code processing: At the end of a DSN session, register 15 contains the highest value placed there by any DSN subcommand used in the session or by any program run by the RUN subcommand. Your runtime environment might format that value as a return code. The value *does not*, however, originate in DSN.

Running a program in TSO foreground
Use the DB2I RUN panel to run a program in TSO foreground. As an alternative to the RUN panel, you can issue the DSN command followed by the RUN subcommand of DSN. Before running the program, be sure to allocate any data sets your program needs.

The following example shows how to start a TSO foreground application. The name of the application is SAMPPGM, and *ssid* is the system ID:

```
TSO Prompt:   READY
Enter:        DSN SYSTEM(ssid)
DSN Prompt:   DSN
Enter:        RUN PROGRAM(SAMPPGM) -
              PLAN(SAMPLAN) -
              LIB(SAMPPROJ.SAMPLIB) -
              PARMS('/D01 D02 D03')
:
 (Here the program runs and might prompt you for input)
DSN Prompt:   DSN
Enter:        END
TSO Prompt:   READY
```

This sequence also works in ISPF option 6. You can package this sequence in a CLIST. DB2 does not support access to multiple DB2 subsystems from a single address space.

The PARMS keyword of the RUN subcommand allows you to pass parameters to the run-time processor and to your application program:

```
PARMS ('/D01, D02, D03')
```

The slash (/) indicates that you are passing parameters. For some languages, you pass parameters and run-time options in the form PARMS('*parameters*/*run-time-options*). In those environments, an example of the PARMS keyword might be:

```
PARMS ('D01, D02, D03/')
```

Check your host language publications for the correct form of the PARMS option.

Running a batch DB2 application in TSO

Most application programs written for the batch environment run under the TSO Terminal Monitor Program (TMP) in background mode. Figure 125 shows the JCL statements you need in order to start such a job. The list that follows explains each statement.

```
//jobname JOB USER=MY DB2ID
//GO EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB DD DSN=prefix.SDSNEXIT,DISP=SHR
//        DD DSN=prefix.SDSNLOAD,DISP=SHR
   .
   .
//SYSTSPRT DD SYSOUT=A
//SYSTSIN DD *
 DSN SYSTEM (ssid)
 RUN PROG (SAMPPGM) -
   PLAN (SAMPLAN) -
   LIB (SAMPPROJ.SAMPLIB) -
   PARMS ('/D01 D02 D03')
 END
 /*
```

Figure 125. Jcl for running a DB2 application under the TSO terminal monitor program

- The JOB option identifies this as a job card. The USER option specifies the DB2 authorization ID of the user.
- The EXEC statement calls the TSO Terminal Monitor Program (TMP).
- The STEPLIB statement specifies the library in which the DSN Command Processor load modules and the default application programming defaults module, DSNHDECP, reside. It can also reference the libraries in which user applications, exit routines, and the customized DSNHDECP module reside. The customized DSNHDECP module is created during installation. If you do not specify a library containing the customized DSNHDECP, DB2 uses the default DSNHDECP.
- Subsequent DD statements define additional files needed by your program.
- The DSN command connects the application to a particular DB2 subsystem.
- The RUN subcommand specifies the name of the application program to run.
- The PLAN keyword specifies plan name.
- The LIB keyword specifies the library the application should access.
- The PARMS keyword passes parameters to the run-time processor and the application program.

- END ends the DSN command processor.

Usage notes:
- Keep DSN job steps short.
- We recommend that you not use DSN to call the EXEC command processor to run CLISTs that contain ISPEXEC statements; results are unpredictable.
- If your program abends or gives you a non-zero return code, DSN terminates.
- You can use a group attachment name instead of a specific *ssid* to connect to a member of a data sharing group. For more information, see *DB2 Data Sharing: Planning and Administration*.

For more information on using the TSO TMP in batch mode, see *OS/390 TSO/E User's Guide*.

Calling applications in a command procedure (CLIST)

As an alternative to the previously described foreground or batch calls to an application, you can also run a TSO or batch application using a command procedure (CLIST).

The following CLIST calls a DB2 application program named MYPROG. The DB2 subsystem name or group attachment name should replace *ssid*.

```
PROC 0                        /* INVOCATION OF DSN FROM A CLIST    */
  DSN SYSTEM(ssid)            /* INVOKE DB2 SUBSYSTEM ssid          */
  IF &LASTCC = 0 THEN         /* BE SURE DSN COMMAND WAS SUCCESSFUL */
    DO                        /* IF SO THEN DO DSN RUN SUBCOMMAND   */
      DATA                    /* ELSE OMIT THE FOLLOWING:           */
        RUN PROGRAM(MYPROG)
        END
      ENDDATA                 /* THE RUN AND THE END ARE FOR DSN    */
    END
EXIT
```

┌─ **IMS** ───┐

To Run a Message-Driven Program

First, be sure you can respond to the program's interactive requests for data and that you can recognize the expected results. Then, enter the transaction code associated with the program. Users of the transaction code must be authorized to run the program.

To run a non-message-driven program

Submit the job control statements needed to run the program.

└───┘

```
  ┌─ CICS ─────────────────────────────────────────────────────────┐
  │  To Run a Program                                                │
  │                                                                  │
  │  First, ensure that the corresponding entries in the RCT, SNT, and RACF* │
  │  control areas allow run authorization for your application. The system  │
  │  administrator is responsible for these functions; see Part 3 (Volume 1) of *DB2* │
  │  *Administration Guide* for more information.                    │
  │                                                                  │
  │  Also, be sure to define to CICS the transaction code assigned to your program │
  │  and the program itself.                                         │
  │                                                                  │
  │  **Make a new copy of the program**                             │
  │                                                                  │
  │  Issue the NEWCOPY command if CICS has not been reinitialized since the │
  │  program was last bound and compiled.                            │
  └──────────────────────────────────────────────────────────────────┘
```

Running a DB2 REXX application

You run DB2 REXX procedures under TSO. You do not precompile, compile,
link-edit or bind DB2 REXX procedures before you run them.

In a batch environment, you might use statements like these to invoke procedure
REXXPROG:

```
//RUNREXX EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSEXEC  DD DISP=SHR,DSN=SYSADM.REXX.EXEC
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
%REXXPROG parameters
```

The SYSEXEC data set contains your REXX application, and the SYSTSIN data set
contains the command that you use to invoke the application.

Using JCL procedures to prepare applications

A number of methods are available for preparing an application to run. You can:
- Use DB2 interactive (DB2I) panels, which lead you step by step through the
  preparation process. See "Using ISPF and DB2 Interactive (DB2I)" on page 431.
- Submit a background job using JCL (which the program preparation panels can
  create for you).
- Start the DSNH CLIST in TSO foreground or background.
- Use TSO prompters and the DSN command processor.
- Use JCL procedures added to your SYS1.PROCLIB (or equivalent) at DB2 install
  time.

This section describes how to use JCL procedures to prepare a program. For
information on using the DSNH CLIST, the TSO DSN command processor, or JCL
procedures added to your SYS1.PROCLIB, see Chapter 2 of *DB2 Command
Reference*.

Available JCL procedures

You can precompile and prepare an application program using a DB2-supplied
procedure. DB2 has a unique procedure for each supported language, with
appropriate defaults for starting the DB2 precompiler and host language compiler or

assembler. The procedures are in *prefix*.SDSNSAMP member DSNTIJMV, which installs the procedures.

Table 53. Procedures for precompiling programs

Language	Procedure	Invocation Included in...
High Level Assembler	DSNHASM	DSNTEJ2A
C	DSNHC	DSNTEJ2D
C++	DSNHCPP DSNHCPP2[2]	DSNTEJ2E N/A
OS/VS COBOL	DSNHCOB	DSNTEJ2C
COBOL/370	DSNHICOB	DSNTEJ2C[1]
COBOL for MVS & VM	DSNHICOB DSNHICB2[2]	DSNTEJ2C[1] N/A
VS COBOL II	DSNHCOB2	DSNTEJ2C[1]
FORTRAN	DSNHFOR	DSNTEJ2F
PL/I	DSNHPLI	DSNTEJ2P
SQL	DSNHSQL	DSNTEJ63

Notes to Table 53
1. You must customize these programs to invoke the procedures listed in this table. For information on how to do that, see Part 2 of *DB2 Installation Guide*.
2. This procedure demonstrates how you can prepare an object-oriented program that consists of two data sets or members, both of which contain SQL.

If you use the PL/I macro processor, you must not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the PARM.PLI= parameter of the EXEC statement in DSNHPLI procedure.

Including code from SYSLIB data sets

To include the proper interface code when you submit the JCL procedures, use one of the sets of statements shown below in your JCL; or, if you are using the call attachment facility, follow the instructions given in "Accessing the CAF language interface" on page 735.

TSO, batch, and CAF

```
//LKED.SYSIN  DD  *
  INCLUDE SYSLIB(member)
/*
```

member must be DSNELI, except for FORTRAN, in which case *member* must be DSNHFT.

```
┌─ IMS ─────────────────────────────────────────────────────────┐
│ //LKED.SYSIN  DD  *                                            │
│   INCLUDE SYSLIB(DFSLI000)                                     │
│   ENTRY (specification)                                        │
│ /*                                                             │
│                                                               │
│ DFSLI000 is the module for DL/I batch attach.                 │
│                                                               │
│ ENTRY specification varies depending on the host language. Include one of │
│ the following:                                                │
│     DLITCBL, for COBOL applications                           │
│     PLICALLA, for PL/I applications                           │
│     Your program's name, for assembler language applications. │
└───────────────────────────────────────────────────────────────┘
```

```
┌─ CICS ────────────────────────────────────────────────────────┐
│ //LKED.SYSIN  DD  *                                            │
│   INCLUDE SYSLIB(DSNCLI)                                       │
│ /*                                                             │
│                                                               │
│ For more information on required CICS modules, see "Step 2: Compile (or │
│ assemble) and link-edit the application" on page 408.         │
└───────────────────────────────────────────────────────────────┘
```

Starting the precompiler dynamically

You can call the precompiler from an assembler program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in *OS/390 MVS Programming: Assembler Services Reference*.

To call the precompiler, specify DSNHPC as the entry point name. You can pass three address options to the precompiler; the following sections describe their formats. The options are addresses of:
- A precompiler option list
- A list of alternate ddnames for the data sets that the precompiler uses
- A page number to use for the first page of the compiler listing on SYSPRINT.

Precompiler option list format
The option list must begin on a two-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list is EBCDIC and can contain precompiler option keywords, separated by one or more blanks, a comma, or both of these.

DDNAME list format
The ddname list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list is an 8-byte field, left-justified, and padded with blanks if needed.

The following table gives the sequence of entries:

Table 54. DDNAME list entries

Entry	Standard ddname	Usage
1	Not applicable	

Table 54. DDNAME list entries (continued)

Entry	Standard ddname	Usage
2	Not applicable	
3	Not applicable	
4	SYSLIB	Library input
5	SYSIN	Source input
6	SYSPRINT	Diagnostic listing
7	Not applicable	
8	SYSUT1	Work data
9	SYSUT2	Work data
10	SYSUT3	Work data
11	Not applicable	
12	SYSTERM	Diagnostic listing
13	Not applicable	
14	SYSCIN	Changed source output
15	Not applicable	
16	DBRMLIB	DBRM output

Page number format

A 6-byte field beginning on a 2-byte boundary contains the page number. The first two bytes must contain the binary value 4 (the length of the remainder of the field). The last 4 bytes contain the page number in characters or zoned decimal.

The precompiler adds 1 to the last page number used in the precompiler listing and puts this value into the page-number field before returning control to the calling routine. Thus, if you call the precompiler again, page numbering is continuous.

An alternative method for preparing a CICS program

<div style="border">

CICS

Instead of using the DB2 Program Preparation panels to prepare your CICS program, you can tailor CICS-supplied JCL procedures to do that. To tailor a CICS procedure, you need to add some steps and change some DD statements. Make changes as needed to do the following:

- Process the program with the DB2 precompiler.
- Bind the application plan. You can do this any time after you precompile the program. You can bind the program either on line by the DB2I panels or as a batch step in this or another MVS job.
- Include a DD statement in the linkage editor step to access the DB2 load library.
- Be sure the linkage editor control statements contain an INCLUDE statement for the DB2 language interface module.

The following example illustrates the necessary changes. This example assumes the use of a VS COBOL II or COBOL/370 program. For any other programming language, change the CICS procedure name and the DB2 precompiler options.

```
         //TESTC01 JOB
         //*
         //*********************************************************
         //*        DB2 PRECOMPILE THE COBOL PROGRAM
         //*********************************************************
 (1)     //PC      EXEC PGM=DSNHPC,
 (1)     //        PARM='HOST(COB2),XREF,SOURCE,FLAG(I),APOST'
 (1)     //STEPLIB  DD  DISP=SHR,DSN=prefix.SDSNEXIT
 (1)     //         DD  DISP=SHR,DSN=prefix.SDSNLOAD
 (1)     //DBRMLIB  DD  DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
 (1)     //SYSCIN   DD  DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
 (1)     //             SPACE=(800,(500,500))
 (1)     //SYSLIB   DD  DISP=SHR,DSN=USER.SRCLIB.DATA
 (1)     //SYSPRINT DD  SYSOUT=*
 (1)     //SYSTERM  DD  SYSOUT=*
 (1)     //SYSUDUMP DD  SYSOUT=*
 (1)     //SYSUT1   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
 (1)     //SYSUT2   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
 (1)     //SYSIN    DD  DISP=SHR,DSN=USER.SRCLIB.DATA(TESTC01)
 (1)     //*
```

</div>

```
┌─ CICS (continued) ─────────────────────────────────────────────────────┐
│          //********************************************************************
│          //***    BIND THIS PROGRAM.
│          //********************************************************************
│   (2)    //BIND    EXEC PGM=IKJEFT01,
│   (2)    //   COND=((4,LT,PC))
│   (2)    //STEPLIB  DD  DISP=SHR,DSN=prefix.SDSNEXIT
│   (2)    //         DD  DISP=SHR,DSN=prefix.SDSNLOAD
│   (2)    //DBRMLIB  DD  DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
│   (2)    //SYSPRINT DD  SYSOUT=*
│   (2)    //SYSTSPRT DD  SYSOUT=*
│   (2)    //SYSUDUMP DD  SYSOUT=*
│   (2)    //SYSTSIN DD *
│   (2)     DSN S(DSN)
│   (2)     BIND PLAN(TESTC01)  MEMBER(TESTC01) ACTION(REP) RETAIN ISOLATION(CS)
│   (2)     END
│          //********************************************************************
│          //*            COMPILE THE COBOL PROGRAM
│          //********************************************************************
│   (3)    //CICS    EXEC DFHEITVL
│   (4)    //TRN.SYSIN    DD  DSN=&&DSNHOUT,DISP=(OLD,DELETE)
│   (5)    //LKED.SYSLMOD DD  DSN=USER.RUNLIB.LOAD
│   (6)    //LKED.DB2LOAD DD  DISP=SHR,DSN=prefix.SDSNLOAD
│          //LKED.SYSIN   DD  *
│   (7)     INCLUDE DB2LOAD(DSNCLI)
│           NAME TESTC01(R)
│          //********************************************************************
│
│  The procedure accounts for these steps:
│     **Step 1.** Precompile the program.
│     **Step 2.** Bind the application plan.
│     **Step 3.** Call the CICS procedure to translate, compile, and link-edit a COBOL program. This
│     procedure has several options you need to consider.
│     **Step 4.** The output of the DB2 precompiler becomes the input to the CICS command language
│     translator.
│     **Step 5.** Reflect an application load library in the data set name of the SYSLMOD DD statement.
│     You must include the name of this load library in the DFHRPL DD statement of the CICS run-time
│     JCL.
│     **Step 6.** Name the DB2 load library that contains the module DSNCLI.
│     **Step 7.** Direct the linkage editor to include the DB2 language interface module for CICS
│     (DSNCLI). In this example, the order of the various control sections (CSECTs) is of no concern
│     because the structure of the procedure automatically satisfies any order requirements.
│
│  For more information about the procedure DFHEITVL, other CICS procedures, or CICS requirements
│  for application programs, please see the appropriate CICS manual.
└────────────────────────────────────────────────────────────────────────┘
```

If you are preparing a particularly large or complex application, you can use one of the last two techniques mentioned above. For example, if your program requires four of your own link-edit include libraries, you cannot prepare the program with DB2I, because DB2I limits the number of include libraries to three plus language, IMS or CICS, and DB2 libraries. Therefore, you would need another preparation method. Programs using the call attachment facility can use either of the last two techniques mentioned above. *Be careful to use the correct language interface.*

Using JCL to prepare a program with object-oriented extensions

If your C⁺⁺ or IBM COBOL for MVS & VM program satisfies both of these conditions, you need special JCL to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

You must precompile the contents of each data set or member separately, but the prelinker must receive all of the compiler output together.

JCL procedures DSNHICB2 and DSNHCPP2, which are in member DSNTIJMV of data set DSN710.SDSNSAMP, show you one way to do this. DSNHICB2 is a procedure for COBOL, and DSNHCPP2 is a procedure for C++.

Using ISPF and DB2 Interactive (DB2I)

If you develop programs using TSO and ISPF, you can prepare them to run using the DB2 Program Preparation panels. These panels guide you step by step through the process of preparing your application to run. There are other ways to prepare a program to run, but using DB2I is the easiest, as it leads you automatically from task to task.

This section describes the options you can specify on the program preparation panels. For the purposes of describing the process, the program preparation examples assume that you are using COBOL programs that run under TSO.

Attention: If your C++ or IBM COBOL for MVS & VM program satisfies both of these conditions, you need to use a JCL procedure to prepare it:
- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

See "Using JCL to prepare a program with object-oriented extensions" on page 430 for more information.

DB2I help

The online help facility enables you to select information in an online DB2 book from a DB2I panel.

For instructions on setting up DB2 online help, see the discussion of setting up DB2 online help in Part 2 of *DB2 Installation Guide*.

If your site makes use of CD-ROM updates, you can make the updated books accessible from DB2I. Select Option 10 on the DB2I Defaults Panel and enter the new book data set names. You must have write access to *prefix*.SDSNCLST to perform this function.

To access DB2I HELP, press PF key 1 (HELP)[1].

The DB2I Primary Option Menu

Figure 126 shows an example of the DB2I Primary Option Menu. From this point, you can access all the DB2I panels without passing through panels that you do not need. To bind a program, enter the number corresponding to BIND/REBIND/FREE to reach the BIND PLAN panel without seeing the ones previous to it.

---

1. Your location could have assigned a different PF key for HELP.

To prepare a new application, beginning with precompilation and working through each of the subsequent preparation steps, begin by entering 3, corresponding to the Program Preparation panel (option 3), as in Figure 126.

```
DSNEPRI                    DB2I PRIMARY OPTION MENU           SSID: DSN
COMMAND ===> 3_


Select one of the following DB2 functions and press ENTER.

 1  SPUFI               (Process SQL statements)
 2  DCLGEN              (Generate SQL and source language declarations)
 3  PROGRAM PREPARATION (Prepare a DB2 application program to run)
 4  PRECOMPILE          (Invoke DB2 precompiler)
 5  BIND/REBIND/FREE    (BIND, REBIND, or FREE plans or packages)
 6  RUN                 (RUN an SQL program)
 7  DB2 COMMANDS        (Issue DB2 commands)
 8  UTILITIES           (Invoke DB2 utilities)
 D  DB2I DEFAULTS       (Set global parameters)
 X  EXIT                (Leave  DB2I)
```

Figure 126. Initiating program preparation through DB2I. Specify Program Preparation on the DB2I Primary Option Menu.

The following explains the functions on the DB2I Primary Option Menu.

1 SPUFI

Lets you develop and execute one or more SQL statements interactively. For further information, see "Chapter 5. Executing SQL from your terminal using SPUFI" on page 51.

2 DCLGEN

Lets you generate C, COBOL, or PL/I data declarations of tables. For further information, see "Chapter 8. Generating declarations for your tables using DCLGEN" on page 95.

3 PROGRAM PREPARATION

Lets you prepare and run an application program to run. For more information, see "The DB2 Program Preparation panel" on page 433.

4 PRECOMPILE

Lets you convert embedded SQL statements into statements that your host language can process. For further information, see "The Precompile panel" on page 440.

5 BIND/REBIND/FREE

Lets you bind, rebind, or free a package or application plan.

6 RUN

Lets you run an application program in a TSO or batch environment.

7 DB2 COMMANDS

Lets you issue DB2 commands. For more information about DB2 commands, see Chapter 2 of *DB2 Command Reference*.

8 UTILITIES

Lets you call DB2 utility programs. For more information, see *DB2 Utility Guide and Reference*.

D DB2I DEFAULTS

Lets you set DB2I defaults. See "DB2I Defaults Panel 1" on page 437.

X EXIT

Lets you exit DB2I.

The DB2 Program Preparation panel

The Program Preparation panel lets you choose whether to perform specific program preparation functions. For the functions you choose, you can also choose whether to display their panels to specify options for performing those functions. Some of the functions you can select are:

- Precompile. The panel for this function lets you control the DB2 precompiler. See page 440.
- Bind a package. The panel for this function lets you bind your program's DBRM to a package (see page 443), and to change your defaults for binding the packages (see page 450).
- Bind a plan. The panel for this function lets you create your program's application plan (see page 447), and to change your defaults for binding the plans (see page 450).
- Compile, link, and run. The panel for these functions let you control the compiler or assembler and the linkage editor. See page 457.

> **TSO and batch**
>
> For TSO programs, you can use the program preparation programs to control the host language run-time processor and the program itself.

The Program Preparation panel also lets you change the DB2I default values (see page 437), and to perform other precompile and prelink functions.

On the DB2 Program Preparation panel, shown in Figure 127, enter the name of the source program data set (this example uses SAMPLEPG.COBOL) and specify the other options you want to include. When finished, press ENTER to view the next panel.

```
DSNEPP01                   DB2 PROGRAM PREPARATION                SSID: DSN
COMMAND ===>_


Enter the following:
 1   INPUT DATA SET NAME .... ===> SAMPLEPG.COBOL
 2   DATA SET NAME QUALIFIER  ===> TEMP       (For building data set names)
 3   PREPARATION ENVIRONMENT  ===> FOREGROUND (FOREGROUND, BACKGROUND, EDITJCL)
 4   RUN TIME ENVIRONMENT ... ===> TSO        (TSO, CAF, CICS, IMS, RRSAF)
 5   OTHER DSNH OPTIONS ..... ===>
                                             (Optional DSNH keywords)
Select functions:          Display panel?       Perform function?
 6   CHANGE DEFAULTS ........ ===> Y  (Y/N)
 7   PL/I MACRO PHASE ....... ===> N  (Y/N)         ===> N  (Y/N)
 8   PRECOMPILE ............. ===> Y  (Y/N)         ===> Y  (Y/N)
 9   CICS COMMAND TRANSLATION                       ===> N  (Y/N)
10   BIND PACKAGE ........... ===> Y  (Y/N)         ===> Y  (Y/N)
11   BIND PLAN.............. ===> Y  (Y/N)         ===> Y  (Y/N)
12   COMPILE OR ASSEMBLE .... ===> Y  (Y/N)         ===> Y  (Y/N)
13   PRELINK................ ===> N  (Y/N)         ===> N  (Y/N)
14   LINK................... ===> N  (Y/N)         ===> Y  (Y/N)
15   RUN.................... ===> N  (Y/N)         ===> Y  (Y/N)
```

Figure 127. The DB2 program preparation panel. Enter the source program data set name and other options.

The following explains the functions on the DB2 Program Preparation panel and how to fill in the necessary fields in order to start program preparation.

1 INPUT DATA SET NAME

> Lets you specify the input data set name. The input data set name can be a PDS or a sequential data set, and can also include a member name. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) qualifies the data set name.
>
> The input data set name you specify is used to precompile, bind, link-edit, and run the program.

2 DATA SET NAME QUALIFIER

> Lets you qualify temporary data set names involved in the program preparation process. Use any character string from 1 to 8 characters that conforms to normal TSO naming conventions. (The default is TEMP.)
>
> For programs that you prepare in the background or that use EDITJCL for the PREPARATION ENVIRONMENT option, DB2 creates a data set named *tsoprefix.qualifier*.CNTL to contain the program preparation JCL. the name *tsoprefix* represents the prefix TSO assigns, and *qualifier* represents the value you enter in the DATA SET NAME QUALIFIER field. If a data set with this name already exists, then DB2 deletes it.

3 PREPARATION ENVIRONMENT

> Lets you specify whether program preparation occurs in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job. Use:
>> FOREGROUND to use the values you specify on the Program Preparation panel and to run immediately.
>> BACKGROUND to create and submit a file containing a DSNH CLIST that runs immediately using the JOB control statement from either the DB2I Defaults panel or your site's SUBMIT exit. The file is saved.
>> EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

4 RUN TIME ENVIRONMENT

> Lets you specify the environment (TSO, CAF, CICS, IMS, RRSAF) in which your program runs.
>
> All programs are prepared under TSO, but can run in any of the environments. If you specify CICS, IMS, or RRSAF, then you must set the RUN field to NO, because you cannot run such programs from the Program Preparation panel. If you set the RUN field to YES, you can specify only TSO or CAF.
>
> (Batch programs also run under the TSO Terminal Monitor Program. You therefore need to specify TSO in this field for batch programs.)

5 OTHER DSNH OPTIONS

> Lets you specify a list of DSNH options that affect the program preparation process, and that override options specified on other panels. If you are using CICS, these can include options you want to specify to the CICS command translator.
>
> If you specify options in this field, separate them by commas. You can continue listing options on the next line, but the total length of the option list can be no more than 70 bytes.
>
> For more information about those options, see DSNH in Chapter 2 of *DB2 Command Reference*.

Fields 7 through 15, described below, let you select the function to perform and to choose whether to show the DB2I panels for the functions you select. Use Y for YES, or N for NO.

If you are willing to accept default values for all the steps, enter N under DISPLAY PANEL for all the other preparation panels listed.

To make changes to the default values, entering Y under DISPLAY PANEL for any panel you want to see. DB2I then displays each of the panels that you request. After all the panels display, DB2 proceeds with the steps involved in preparing your program to run.

Variables for all functions used during program preparation are maintained separately from variables entered from the DB2I Primary Option Menu. For example, the bind plan variables you enter on the program preparation panel are saved separately from those on any bind plan panel that you reach from the Primary Option Menu.

6 CHANGE DEFAULTS

Lets you specify whether to change the DB2I defaults. Enter Y in the *Display Panel* field next to this option; otherwise enter N. Minimally, you should specify your subsystem identifier and programming language on the defaults panel. For more information, see "DB2I Defaults Panel 1" on page 437.

7 PL/I MACRO PHASE

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel to control the PL/I macro phase by entering PL/I options in the OPTIONS field of that panel. That panel also displays for options COMPILE OR ASSEMBLE, LINK, and RUN.

This field applies to PL/I programs only. If your program is not a PL/I program or does not use the PL/I macro processor, specify N in the *Perform function* field for this option, which sets the *Display panel* field to the default N.

For information on PL/I options, see "The Program Preparation: Compile, Link, and Run panel" on page 457.

8 PRECOMPILE

Lets you specify whether to display the Precompile panel. To see this panel enter Y in the *Display Panel* field next to this option; otherwise enter N. For information on the Precompile panel, see "The Precompile panel" on page 440.

9 CICS COMMAND TRANSLATION

Lets you specify whether to use the CICS command translator. This field applies to CICS programs only.

IMS and TSO

If you run under TSO or IMS, ignore this step; this allows the *Perform function* field to default to N.

```
┌─ CICS ─────────────────────────────────────────────────────────┐
│                                                                 │
│  If you are using CICS and have precompiled your program, you must │
│  translate your program using the CICS command translator.      │
│                                                                 │
│  There is no separate DB2I panel for the command translator. You can │
│  specify translation options on the *Other Options* field of the DB2 │
│  Program Preparation panel, or in your source program if it is not an │
│  assembler program.                                             │
│                                                                 │
│  Because you specified a CICS run-time environment, the *Perform* │
│  *function* column defaults to Y. Command translation takes place │
│  automatically after you precompile the program.                │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

10 BIND PACKAGE

Lets you specify whether to display the BIND PACKAGE panel. To see it, enter *Y* in the *Display panel* field next to this option; otherwise, enter N. For information on the panel, see "The Bind Package panel" on page 443.

11 BIND PLAN

Lets you specify whether to display the BIND PLAN panel. To see it, enter *Y* in the *Display panel* field next to this option; otherwise, enter N. For information on the panel, see "The Bind Plan panel" on page 447.

12 COMPILE OR ASSEMBLE

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see this panel enter Y in the *Display Panel* field next to this option; otherwise, enter N.

For information on the panel, see "The Program Preparation: Compile, Link, and Run panel" on page 457.

13 PRELINK

Lets you use the prelink utility to make your C, C++, or IBM COBOL for MVS & VM program reentrant. This utility concatenates compile-time initialization information from one or more text decks into a single initialization unit. To use the utility, enter Y in the *Display Panel* field next to this option; otherwise, enter N. If you request this step, then you must also request the compile step and the link-edit step.

For more information on the prelink utility, see *OS/390 Language Environment for OS/390 & VM Programming Guide*.

14 LINK

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see it, enter Y in the *Display Panel* field next to this option; otherwise, enter N. If you specify Y in the *Display Panel* field for the COMPILE OR ASSEMBLE option, you do not need to make any changes to this field; the panel displayed for COMPILE OR ASSEMBLE is the same as the panel displayed for LINK. You can make the changes you want to affect the link-edit step at the same time you make the changes to the compile step.

For information on the panel, see "The Program Preparation: Compile, Link, and Run panel" on page 457.

15 RUN

Lets you specify whether to run your program. The RUN option is available only if you specify TSO or CAF for RUN TIME ENVIRONMENT.

If you specify Y in the *Display Panel* field for the COMPILE OR ASSEMBLE or LINK option, you can specify N in this field, because the panel displayed for COMPILE OR ASSEMBLE and for LINK is the same as the panel displayed for RUN.

> **IMS and CICS**
>
> IMS and CICS programs cannot run using DB2I. If you are using IMS or CICS, use N in these fields.

> **TSO and batch**
>
> If you are using TSO and want to run your program, you must enter Y in the *Perform function* column next to this option. You can also indicate that you want to specify options and values to affect the running of your program, by entering Y in the *Display panel* column.
>
> For information on the panel, see "The Program Preparation: Compile, Link, and Run panel" on page 457.

Pressing ENTER takes you to the first panel in the series you specified, in this example to the DB2I Defaults panel. If, at any point in your progress from panel to panel, you press the END key, you return to this first panel, from which you can change your processing specifications. Asterisks (*) in the *Display Panel* column of rows 7 through 14 indicate which panels you have already examined. You can see a panel again by writing a Y over an asterisk.

DB2I Defaults Panel 1

DB2I Defaults panel 1 lets you change many of the system defaults set at DB2 install time. Figure 128 shows the fields that affect the processing of the other DB2I panels.

```
DSNEOP01              DB2I DEFAULTS PANEL 1
COMMAND ===>_

Change defaults as desired:

 1  DB2 NAME ............. ===> DSN        (Subsystem identifier)
 2  DB2 CONNECTION RETRIES ===> 0          (How many retries for DB2 connection)
 3  APPLICATION LANGUAGE   ===> COBOL      (ASM, C, CPP, COBOL, COB2, IBMCOB,
                                            FORTRAN,PLI)
 4  LINES/PAGE OF LISTING  ===> 60         (A number from 5 to 999)
 5  MESSAGE LEVEL ........ ===> I          (Information, Warning, Error, Severe)
 6  SQL STRING DELIMITER   ===> DEFAULT    (DEFAULT, ' or ")
 7  DECIMAL POINT ........ ===> .          (. or ,)
 8  STOP IF RETURN CODE >= ===> 8          (Lowest terminating return code)
 9  NUMBER OF ROWS         ===> 20         (For ISPF Tables)
10  CHANGE HELP BOOK NAMES?===> NO         (YES to change HELP data set names)
```

Figure 128. DB2I defaults panel 1

The following explains the fields on DB2I Defaults panel 1.

1 DB2 NAME

Lets you specify the DB2 subsystem that processes your DB2I requests. If you specify a different DB2 subsystem, its identifier displays in the SSID (subsystem identifier) field located at the top, right side of your screen. The default is DSN.

2 DB2 CONNECTION RETRIES

Lets you specify the number of additional times to attempt to connect to DB2, if DB2 is not up when the program issues the DSN command. The program preparation process does not use this option.

Use a number from 0 to 120. The default is 0. Connections are attempted at 30-second intervals.

3 APPLICATION LANGUAGE

Lets you specify the default programming language for your application program. You can specify any of the following:

ASM

For High Level Assembler/MVS

C   For C/370

CPP

For C++

COBOL

For OS/VS COBOL (default)

COB2

For VS COBOL II

IBMCOB

For IBM SAA AD/Cycle COBOL/370 or IBM COBOL for MVS & VM

FORTRAN

For VS FORTRAN

PLI

For PL/I

If you specify COBOL, COB2, or IBMCOB, DB2 prompts you for more COBOL defaults on panel DSNEOP02. See "DB2I Defaults Panel 2".

You cannot specify FORTRAN for IMS or CICS programs.

4 LINES/PAGE OF LISTING

Lets you specify the number of lines to print on each page of listing or SPUFI output. The default is 60.

5 MESSAGE LEVEL

Lets you specify the lowest level of message to return to you during the BIND phase of the preparation process. Use:
I   For all information, warning, error, and severe error messages
W   For warning, error, and severe error messages
E   For error and severe error messages
S   For severe error messages only

6 SQL STRING DELIMITER

Lets you specify the symbol used to delimit a string in SQL statements in COBOL programs. This option is valid only when the application language is COBOL, COB2, or IBMCOB. Use:
DEFAULT
    To use the default defined at install time
'   For an apostrophe
"   For a quotation mark

7 DECIMAL POINT

Lets you specify how your host language source program represents decimal separators and how SPUFI displays decimal separators in its output. Use a comma (,) or a period (.). The default is a period (.).

8 STOP IF RETURN CODE >=

Lets you specify the smallest value of the return code (from precompile, compile, link-edit, or bind) that will prevent later steps from running. Use:
4   To stop on warnings and more severe errors.
8   To stop on errors and more severe errors. The default is 8.

9 NUMBER OF ROWS

Lets you specify the default number of input entry rows to generate on the initial display of ISPF panels. The number of rows with non-blank entries determines the number of rows that appear on later displays.

10 CHANGE HELP BOOK NAMES?

Lets you change the name of the BookManager® book you reference for online help. The default is NO.

Suppose that the default programming language is PL/I and the default number of lines per page of program listing is 60. Your program is in COBOL, so you want to change field 3, APPLICATION LANGUAGE. You also want to print 80 lines to the page, so you need to change field 4, LINES/PAGE OF LISTING, as well. Figure 128 on page 438 shows the entries that you make in DB2I Defaults panel 1 to make these changes. In this case, pressing ENTER takes you to DB2 DEFAULTS panel 2.

DB2I Defaults Panel 2

After you press Enter on the DB2I DEFAULTS panel 1, the DB2I DEFAULTS panel 2 is displayed. If you chose COBOL, COB2, or IBMCOB as the language on the DB2I Defaults panel 1, three fields are displayed. Otherwise, only the first field is

displayed. Figure 129 shows the DB2I DEFAULTS panel 2 when COBOL, COB2, or
IBMCOB is selected.

```
DSNEOP02              DB2I DEFAULTS PANEL 2
COMMAND ===>_

Change defaults as desired:

 1  DB2I JOB STATEMENT:   (Optional if your site has a SUBMIT exit)
    ===> //USRT001A JOB (ACCOUNT),'NAME'
    ===> //*
    ===> //*
    ===> //*

    COBOL DEFAULTS:                       (For COBOL, COB2, or IBMCOB)
 2  COBOL STRING DELIMITER ===> DEFAULT   (DEFAULT, ' or ")
 3  DBCS SYMBOL FOR DCLGEN ===> G         (G/N - Character in PIC clause)
```

Figure 129. DB2I defaults panel 2

1 DB2I JOB STATEMENT

Lets you change your default job statement. Specify a job control
statement, and optionally, a JOBLIB statement to use either in the
background or the EDITJCL program preparation environment. Use a
JOBLIB statement to specify run-time libraries that your application
requires. If your program has a SUBMIT exit routine, DB2 uses that routine.
If that routine builds a job control statement, you can leave this field blank.

2 COBOL STRING DELIMITER

Lets you specify the symbol used to delimit a string in a COBOL statement
in a COBOL application. Use:

DEFAULT

To use the default defined at install time

'   For an apostrophe

"   For a quotation mark

Leave this field blank to accept the default value.

3 DBCS SYMBOL FOR DCLGEN

Lets you enter either G (the default) or N, to specify whether DCLGEN
generates a picture clause that has the form PIC G(*n*) DISPLAY-1 or PIC
N(*n*).

Leave this field blank to accept the default value.

Pressing ENTER takes you to the next panel you specified on the DB2 Program
Preparation panel, in this case, to the Precompile panel.

The Precompile panel

The next step in the process is to precompile. Figure 126 on page 432, the DB2I
Primary Option Menu, shows that you can reach the Precompile panel in two ways:
you can either specify it as a part of the program preparation process from the DB2
Program Preparation panel, or you can reach it directly from the DB2I Primary
Option Menu. The way you choose to reach the panel determines the default values
of the fields it contains. Figure 130 on page 441 shows the Precompile panel.

```
DSNETP01                    PRECOMPILE                      SSID: DSN
COMMAND ===>_

Enter precompiler data sets:
 1  INPUT DATA SET .... ===> SAMPLEPG.COBOL
 2  INCLUDE LIBRARY ... ===> SRCLIB.DATA

 3  DSNAME QUALIFIER .. ===> TEMP          (For building data set names)
 4  DBRM DATA SET ..... ===>

Enter processing options as desired:
 5  WHERE TO PRECOMPILE ===> FOREGROUND    (FOREGROUND, BACKGROUND, or EDITJCL)
 6  VERSION ........... ===>
                                           (Blank, VERSION, or AUTO)
 7  OTHER OPTIONS ..... ===>
```

Figure 130. The precompile panel. Specify the include library, if any, that your program should use, and any other options you need.

The following explain the functions on the Precompile panel, and how to enter the fields for preparing to precompile.

1 INPUT DATA SET

Lets you specify the data set name of the source program and SQL statements to precompile.

If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name specified there. You can override it on this panel if you wish.

If you reached this panel directly from the DB2I Primary Option Menu, you must enter the data set name of the program you want to precompile. The data set name can include a member name. If you do not enclose the data set name with apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

2 INCLUDE LIBRARY

Lets you enter the name of a library containing members that the precompiler should include. These members can contain output from DCLGEN. If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) qualifies the name.

You can request additional INCLUDE libraries by entering DSNH CLIST parameters of the form P*n*LIB(*dsname*), where *n* is 2, 3, or 4) on the OTHER OPTIONS field of this panel or on the OTHER DSNH OPTIONS field of the Program Preparation panel.

3 DSNAME QUALIFIER

Lets you specify a character string that qualifies temporary data set names during precompile. Use any character string from 1 to 8 characters in length that conforms to normal TSO naming conventions.

If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name qualifier specified there. You can override it on this panel if you wish.

If you reached this panel from the DB2I Primary Option Menu, you can either specify a DSNAME QUALIFIER or let the field take its default value, TEMP.

> **IMS and TSO**
>
> For IMS and TSO programs, DB2 stores the precompiled source
> statements (to pass to the compile or assemble step) in a data set
> named *tsoprefix.qualifier.suffix*. A data set named
> *tsoprefix.qualifier*.PCLIST contains the precompiler print listing.
>
> For programs prepared in the background or that use the
> PREPARATION ENVIRONMENT option EDITJCL (on the DB2
> Program Preparation panel), a data set named *tsoprefix.qualifier*.CNTL
> contains the program preparation JCL.
>
> In these examples, *tsoprefix* represents the prefix TSO assigns, often
> the same as the authorization ID. *qualifier* represents the value
> entered in the DSNAME QUALIFIER field. And *suffix* represents the
> output name, which is one of the following: COBOL, FORTRAN, C,
> PLI, ASM, DECK, CICSIN, OBJ, or DATA. In the example in
> Figure 130 on page 441, the data set *tsoprefix*.TEMP.COBOL contains
> the precompiled source statements, and *tsoprefix*.TEMP.PCLIST
> contains the precompiler print listing. If data sets with these names
> already exist, then DB2 deletes them.

> **CICS**
>
> For CICS programs, the data set *tsoprefix.qualifier.suffix* receives the
> precompiled source statements in preparation for CICS command
> translation.
>
> If you do not plan to do CICS command translation, the source
> statements in *tsoprefix.qualifier.suffix*, are ready to compile. The data
> set *tsoprefix.qualifier*.PCLIST contains the precompiler print listing.
>
> When the precompiler completes its work, control passes to the CICS
> command translator. Because there is no panel for the translator,
> translation takes place automatically. The data set
> *tsoprefix.qualifier*.CXLIST contains the output from the command
> translator.

4 DBRM DATA SET

Lets you name the DBRM library data set for the precompiler output. The
data set can also include a member name.

When you reach this panel, the field is blank. When you press ENTER,
however, the value contained in the DSNAME QUALIFIER field of the
panel, concatenated with *DBRM*, specifies the DBRM data set:
qualifier.DBRM.

You can enter another data set name in this field only if you allocate and
catalog the data set before doing so. This is true even if the data set name
that you enter corresponds to what is otherwise the default value of this
field.

The precompiler sends modified source code to the data set *qualifier.host*, where *host* is the language specified in the APPLICATION LANGUAGE field of DB2I Defaults panel 1.

5 WHERE TO PRECOMPILE

Lets you indicate whether to precompile in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job.

If you reached this panel from the DB2 Program Preparation panel, the field contains the preparation environment specified there. You can override that value if you wish.

If you reached this panel directly from the DB2I Primary Option Menu, you can either specify a processing environment or allow this field to take its default value. Use:

FOREGROUND to immediately precompile the program with the values you specify in these panels.

BACKGROUND to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either DB2I Defaults panel 2 or your site's SUBMIT exit. The file is saved.

EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

6 VERSION

Lets you specify the version of the program and its DBRM. If the version contains the maximum number of characters permitted (64), you must enter each character with no intervening blanks from one line to the next. This field is optional.

See "Advantages of packages" on page 316 for more information about this option.

7 OTHER OPTIONS

Lets you enter any option that the DSNH CLIST accepts, which gives you greater control over your program. The DSNH options you specify in this field override options specified on other panels. The option list can continue to the next line, but the total length of the list can be no more than 70 bytes.

For more information on DSNH options, see Chapter 2 of *DB2 Command Reference*.

The Bind Package panel

When you request this option, the panel displayed is the first of two BIND PACKAGE panels. You can reach the BIND PACKAGE panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. If you enter the BIND PACKAGE panel from the Program Preparation panel, many of the BIND PACKAGE entries contain values from the Primary and Precompile panels. Figure 131 shows the BIND PACKAGE panel.

```
DSNEBP07                    BIND PACKAGE                    SSID: DSN

 COMMAND ===>_

 Specify output location and collection names:
  1  LOCATION NAME ............. ===>                    (Defaults to local)
  2  COLLECTION-ID ............. ===>                    (Required)
 Specify package source (DBRM or COPY):
  3   DBRM:       COPY:    ===> DBRM      (Specify DBRM or COPY)
  4  MEMBER   or  COLLECTION-ID  ===>
  5  PASSWORD or  PACKAGE-ID ..  ===>
  6  LIBRARY  or  VERSION .....  ===>
                                                   (Blank, or COPY version-id)
  7  ........ --  OPTIONS .....  ===>              (COMPOSITE or COMMAND)
 Enter options as desired:
  8  CHANGE CURRENT DEFAULTS?    ===> NO      (NO or YES)
  9  ENABLE/DISABLE CONNECTIONS? ===> NO      (NO or YES)
 10  OWNER OF PACKAGE (AUTHID).. ===>         (Leave blank for primary ID)
 11  QUALIFIER ................. ===>         (Leave blank for OWNER)
 12  ACTION ON PACKAGE ........  ===> REPLACE (ADD or REPLACE)
 13  INCLUDE PATH? ............. ===> NO      (NO or YES)
 14  REPLACE VERSION ..........  ===>         (Replacement version-id)
 15  IMMEDIATE WRITE OPTION ... ===> NO       (NO, YES, PH1)
```

Figure 131. The bind package panel

The following information explains the functions on the BIND PACKAGE panel and how to fill the necessary fields in order to bind your program. For more information, see the BIND PACKAGE command in Chapter 2 of *DB2 Command Reference*.

1 LOCATION NAME

> Lets you specify the system at which to bind the package. You can use from 1 to 16 characters to specify the location name. The location name must be defined in the catalog table SYSIBM.LOCATIONS. The default is the local DBMS.

2 COLLECTION-ID

> Lets you specify the collection the package is in. You can use from 1 to 18 characters to specify the collection, and the first character must be alphabetic.

3 DBRM: COPY:

> Lets you specify whether you are creating a new package (DBRM) or making a copy of a package that already exists (COPY). Use:
>
> **DBRM**
>
> > To create a new package. You must specify values in the LIBRARY, PASSWORD, and MEMBER fields.
>
> **COPY**
>
> > To copy an existing package. You must specify values in the COLLECTION-ID and PACKAGE-ID fields. (The VERSION field is optional.)

4 MEMBER or COLLECTION-ID

> **MEMBER (for new packages):** If you are creating a new package, this option lets you specify the DBRM to bind. You can specify a member name from 1 to 8 characters. The default name depends on the input data set name.
>
> • If the input data set is partitioned, the default name is the member name of the input data set specified in the INPUT DATA SET NAME field of the DB2 Program Preparation panel.

- If the input data set is sequential, the default name is the second qualifier of this input data set.

COLLECTION-ID (for copying a package): If you are copying a package, this option specifies the collection ID that contains the original package. You can specify a collection ID from 1 to 18 characters, which must be different from the collection ID specified on the PACKAGE ID field.

5 PASSWORD or PACKAGE-ID

PASSWORD (for new packages): If you are creating a new package, this lets you enter password for the library you list in the LIBRARY field. You can use this field only if you reached the BIND PACKAGE panel directly from the DB2 Primary Option Menu.

PACKAGE-ID (for copying packages): If you are copying a package, this option lets you specify the name of the original package. You can enter a package ID from 1 to 8 characters.

6 LIBRARY or VERSION

LIBRARY (for new packages): If you are creating a new package, this lets you specify the names of the libraries that contain the DBRMs specified on the MEMBER field for the bind process. Libraries are searched in the order specified and must in the catalog tables.

VERSION (for copying packages): If you are copying a package, this option lets you to specify the version of the original package. You can specify a version ID from 1 to 64 characters. See "Advantages of packages" on page 316 for more information about this option.

7 OPTIONS

Lets you specify which bind options DB2 uses when you issue BIND PACKAGE with the COPY option. Specify:

COMPOSITE (default) to cause DB2 to use any options you specify in the BIND PACKAGE command. For all other options, DB2 uses the options of the copied package.

COMMAND to cause DB2 to use the options you specify in the BIND PACKAGE command. For all other options, DB2 uses the following values:

– For a local copy of a package, DB2 uses the defaults for the local DB2 subsystem.

– For a remote copy of a package, DB2 uses the defaults for the server on which the package is bound.

8 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding packages. If you enter YES in this field, you see the Defaults for BIND PACKAGE panel as your next step. You can enter your new preferences there; for instructions, see "The Defaults for Bind or Rebind Package or Plan panels" on page 450.

9 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 135 on page 455) that lets you specify whether various system connections are valid for this application. You can specify connection names to further

identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

10 OWNER OF PACKAGE (AUTHID)

Lets you specify the primary authorization ID of the owner of the new package. That ID is the name owning the package, and the name associated with all accounting and trace records produced by the package.

The owner must have the privileges required to run SQL statements contained in the package.

The default is the primary authorization ID of the bind process.

11 QUALIFIER

Lets you specify the implicit qualifier for unqualified tables, views, indexes, and aliases. You can specify a qualifier from 1 to 8 characters. The default is the authorization ID of the package owner.

12 ACTION ON PACKAGE

Lets you specify whether to replace an existing package or create a new one. Use:

REPLACE (default) to replace the package named in the PACKAGE-ID field if it already exists, and add it if it does not. (Use this option if you are changing the package because the SQL statements in the program changed. If only the SQL environment changes but not the SQL statements, you can use REBIND PACKAGE.)
ADD to add the package named in the PACKAGE-ID field, only if it does not already exist.

13 INCLUDE PATH?

Indicates whether you will supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

14 REPLACE VERSION

Lets you specify whether to replace a specific version of an existing package or create a new one. If the package and the version named in the PACKAGE-ID and VERSION fields already exist, you must specify REPLACE. You can specify a version ID from 1 to 64 characters. The default version ID is that specified in the VERSION field.

15 IMMEDIATE WRITE OPTION

Specifies when DB2 writes the changes for updated group buffer pool-dependent pages. This field applies only to a data sharing environment. The values that you can specify are:

NO   For transactions that are committed, write the changes at or before phase 2 of the commit process. For transactions that are rolled back, write the changes at the end of the abort process. NO is the default.

YES   Write the changes immediately after group buffer pool-dependent pages are updated.

PH1   Write the changes at or before phase 1 of the commit process. If the transaction is rolled back later, write the additional changes that are caused by the rollback at the end of the abort process.

The Bind Plan panel

The BIND PLAN panel is the first of two BIND PLAN panels. It specifies options in the bind process of an application plan. Like the Precompile panel, you can reach the BIND PLAN panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. You must have an application plan, even if you bind your application to packages; this panel also follows the BIND PACKAGE panels.

If you enter the BIND PLAN panel from the Program Preparation panel, many of the BIND PLAN entries contain values from the Primary and Precompile panels. See Figure 132.

```
DSNEBP02                    BIND PLAN                      SSID: DSN
 COMMAND ===>_

 Enter DBRM data set name(s):
  1  MEMBER .......... ===> SAMPLEPG
  2  PASSWORD ........ ===>
  3  LIBRARY ......... ===> TEMP.DBRM
  4  ADDITIONAL DBRMS? ........ ===> NO        (YES to list more DBRMs)

 Enter options as desired:
  5  PLAN NAME ................ ===> SAMPLEPG  (Required to create a plan)
  6  CHANGE CURRENT DEFAULTS?   ===> NO        (NO or YES)
  7  ENABLE/DISABLE CONNECTIONS?===> NO        (NO or YES)
  8  INCLUDE PACKAGE LIST?..... ===> NO        (NO or YES)
  9  OWNER OF PLAN (AUTHID) ... ===>           (Leave blank for your primary ID)
 10  QUALIFIER ................ ===>           (For tables, views, and aliases)
 11  CACHESIZE ................ ===> 0         (Blank, or value 0-4096)
 12  ACTION ON PLAN ........... ===> REPLACE   (REPLACE or ADD)
 13  RETAIN EXECUTION AUTHORITY ===> YES       (YES to retain user list)
 14  CURRENT SERVER ........... ===>                   (Location name)
 15  INCLUDE PATH? ............ ===>           (NO or YES)
 16  IMMEDIATE WRITE OPTION ... ===> NO        (NO, YES, PH1)
```

Figure 132. The bind plan panel

The following explains the functions on the BIND PLAN panel and how to fill the necessary fields in order to bind your program. For more information, see the BIND PLAN command in Chapter 2 of *DB2 Command Reference*.

1 MEMBER

Lets you specify the DBRMs to include in the plan. You can specify a name from 1 to 8 characters. You must specify MEMBER or INCLUDE PACKAGE LIST, or both. If you do not specify MEMBER, fields 2, 3, and 4 are ignored.

The default member name depends on the input data set.

- If the input data set is partitioned, the default name is the member name of the input data set specified in field 1 of the DB2 Program Preparation panel.

- If the input data set is sequential, the default name is the second qualifier of this input data set.

If you reached this panel directly from the DB2I Primary Option Menu, you must provide values for the MEMBER and LIBRARY fields.

If you plan to use more than one DBRM, you can include the library name and member name of each DBRM in the MEMBER and LIBRARY fields, separating entries with commas. You can also specify more DBRMs by using the ADDITIONAL DBRMS? field on this panel.

2 PASSWORD

Lets you enter passwords for the libraries you list in the LIBRARY field. You can use this field only if you reached the BIND PLAN panel directly from the DB2 Primary Option Menu.

3 LIBRARY

Lets you specify the name of the library or libraries that contain the DBRMs to use for the bind process. You can specify a name up to 44 characters long.

4 ADDITIONAL DBRMS?

Lets you specify more DBRM entries if you need more room. Or, if you reached this panel as part of the program preparation process, you can include more DBRMs by entering YES in this field. A separate panel then displays, where you can enter more DBRM library and member names; see "Panels for entering lists of values" on page 456.

5 PLAN NAME

Lets you name the application plan to create. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table.

If you reached this panel through the DB2 Program Preparation panel, the default for this field depends on the value you entered in the INPUT DATA SET NAME field of that panel.

If you reached this panel directly from the DB2 Primary Option Menu, you must include a plan name if you want to create an application plan. The default name for this field depends on the input data set:

- If the input data set is partitioned, the default name is the member name.
- If the input data set is sequential, the default name is the second qualifier of the data set name.

6 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding plans. If you enter YES in this field, you see the Defaults for BIND PLAN panel as your next step. You can enter your new preferences there; for instructions, see "The Defaults for Bind or Rebind Package or Plan panels" on page 450.

7 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 135 on page 455) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

8 INCLUDE PACKAGE LIST?

Lets you include a list of packages in the plan. If you specify YES, a separate panel displays on which you must enter the package location,

collection name, and package name for each package to include in the plan (see "Panels for entering lists of values" on page 456). This list is optional if you use the MEMBER field.

You can specify a location name from 1 to 16 characters, a collection ID from 1 to 18 characters, and a package ID from 1 to 8 characters. If you specify a location name, which is optional, it must be in the catalog table SYSIBM.LOCATIONS; the default location is the local DBMS.

You must specify INCLUDE PACKAGE LIST? or MEMBER, or both, as input to the bind plan.

9 OWNER OF PLAN (AUTHID)

Lets you specify the primary authorization ID of the owner of the new plan. That ID is the name owning the plan, and the name associated with all accounting and trace records produced by the plan.

The owner must have the privileges required to run SQL statements contained in the plan.

10 QUALIFIER

Lets you specify the implicit qualifier for unqualified tables, views and aliases. You can specify a name from 1 to 8 characters, which must conform to the rules for SQL short identifiers. If you leave this field blank, the default qualifier is the authorization ID of the plan owner.

11 CACHESIZE

Lets you specify the size (in bytes) of the authorization cache. Valid values are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that DB2 does not use an authorization cache. The default is 1024.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead. See "Determining the optimal authorization cache size" on page 417 for more information about this option.

12 ACTION ON PLAN

Lets you specify whether this is a new or changed application plan. Use:
REPLACE (default) to replace the plan named in the PLAN NAME field if it already exists, and add the plan if it does not exist.
ADD to add the plan named in the PLAN NAME field, only if it does not already exist.

13 RETAIN EXECUTION AUTHORITY

Lets you choose whether or not those users with the authority to bind or run the existing plan are to keep that authority over the changed plan. This applies only when you are replacing an existing plan.

If the plan ownership changes and you specify YES, the new owner grants BIND and EXECUTE authority to the previous plan owner.

If the plan ownership changes and you do not specify YES, then everyone but the new plan owner loses EXECUTE authority (but not BIND authority), and the new plan owner grants BIND authority to the previous plan owner.

14 CURRENT SERVER

Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSIBM.LOCATIONS.

If you specify a remote server, DB2 connects to that server when the first SQL statement executes. The default is the name of the local DB2 subsystem. For more information about this option, see the bind option CURRENTSERVER in Chapter 2 of *DB2 Command Reference*.

15 INCLUDE PATH?
Indicates whether you will supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

16 IMMEDIATE WRITE OPTION
Specifies when DB2 writes the changes for updated group buffer pool-dependent pages. This field applies only to a data sharing environment. The values that you can specify are:

NO For transactions that are committed, write the changes at or before phase 2 of the commit process. For transactions that are rolled back, write the changes at the end of the abort process. NO is the default.

YES Write the changes immediately after group buffer pool-dependent pages are updated.

PH1 Write the changes at or before phase 1 of the commit process. If the transaction is rolled back later, write the additional changes that are caused by the rollback at the end of the abort process.

For more information about this option, see the bind option IMMEDWRITE in Chapter 2 of *DB2 Command Reference*.

When you finish making changes to this panel, press ENTER to go to the second of the program preparation panels, Program Prep: Compile, Link, and Run.

The Defaults for Bind or Rebind Package or Plan panels

On this panel, enter new defaults for binding the package.

```
DSNEBP10                   DEFAULTS FOR BIND PACKAGE              SSID: DSN
 COMMAND ===> _

 Change default options as necessary:

  1  ISOLATION LEVEL ......... ===>           (RR, RS, CS, UR, or NC)
  2  VALIDATION TIME ......... ===>           (RUN or BIND)
  3  RESOURCE RELEASE TIME ... ===>           (COMMIT or DEALLOCATE)
  4  EXPLAIN PATH SELECTION .. ===>           (NO or YES)
  5  DATA CURRENCY ........... ===>           (NO or YES)
  6  PARALLEL DEGREE ......... ===>           (1 or ANY)
  7  SQLERROR PROCESSING ..... ===>           (NOPACKAGE or CONTINUE)
  8  REOPTIMIZE FOR INPUT VARS ===>           (NO OR YES)
  9  DEFER PREPARE ........... ===>           (NO OR YES)
 10  KEEP DYN SQL PAST COMMIT  ===>           (NO or YES)
 11  DBPROTOCOL .............. ===>           (DRDA OR PRIVATE)
 12  APPLICATION ENCODING ...  ===>           (Blank, ASCII, EBCDIC,
                                                 UNICODE, or ccsid)
 13  OPTIMIZATION HINT ......  ===>           (Blank or 'hint-id')
 14  IMMEDIATE WRITE ......... ===> NO        (YES, NO, PH1)
 15  DYNAMIC RULES ........... ===>           (RUN, BIND,
                                                   DEFINERUN, DEFINEBIND,
                                                   INVOKERUN, INVOKEBIND)
```

Figure 133. The defaults for bind package panel

This panel lets you change your defaults for BIND PACKAGE options. With a few
minor exceptions, the options on this panel are the same as the options for the
defaults for rebinding a package. However, the defaults for REBIND PACKAGE are
different from those shown in the above figure, and you can specify SAME in any
field to specify the values used the last time the package was bound. For rebinding,
the default value for all fields is SAME.

On this panel, enter new defaults for binding your plan.

```
DSNEBP10                   DEFAULTS FOR BIND PLAN                 SSID: DSN
 COMMAND ===>

 Change default options as necessary:

  1  ISOLATION LEVEL ......... ===> RR        (RR, RS, CS, or UR)
  2  VALIDATION TIME ......... ===> RUN       (RUN or BIND)
  3  RESOURCE RELEASE TIME ... ===> COMMIT    (COMMIT or DEALLOCATE)
  4  EXPLAIN PATH SELECTION .. ===> NO        (NO or YES)
  5  DATA CURRENCY ........... ===> NO        (NO or YES)
  6  PARALLEL DEGREE ......... ===> 1         (1 or ANY)
  7  RESOURCE ACQUISITION TIME ===> USE       (USE or ALLOCATE)
  8  REOPTIMIZE FOR INPUT VARS ===> NO        (NO OR YES)
  9  DEFER PREPARE ........... ===> NO        (NO or YES)
 10  KEEP DYN SQL PAST COMMIT. ===> NO        (NO or YES)
 11  DBPROTOCOL .............. ===>           (Blank, DRDA, OR PRIVATE)
 12  APPLICATION ENCODING ...  ===>           (Blank, ASCII, EBCDIC,
                                                 UNICODE, or ccsid)
 13  OPTIMIZATION HINT ......  ===>           (Blank or 'hint-id')
 14  IMMEDIATE WRITE ......... ===> NO        (YES, NO, PH1)
 15  DYNAMIC RULES ........... ===> RUN       (RUN or BIND)
 16  SQLRULES................. ===> DB2       (DB2 or STD)
 17  DISCONNECT .............. ===> EXPLICIT  (EXPLICIT, AUTOMATIC,
                                              or CONDITIONAL)
```

Figure 134. The defaults for bind plan panel

This panel lets you change your defaults for options of BIND PLAN. The options on this panel are mostly the same as the options for the defaults for rebinding a package. However, for REBIND PLAN defaults, you can specify SAME in any field to specify the values used the last time the plan was bound. For rebinding, the default value for all fields is SAME.

Explanations of panel fields: The fields in panels DEFAULTS FOR BIND PACKAGE and DEFAULTS FOR BIND PLAN are:

1 ISOLATION LEVEL

Lets you specify how far to isolate your application from the effects of other running applications. The default is the value used for the old plan or package if you are replacing an existing one.

Use RR, RS, CS, or UR. For a description of the effects of those values, see "The ISOLATION option" on page 341.

2 VALIDATION TIME

Lets you specify RUN or BIND to tell whether to check authorization at run time or at bind time. The default is that used for the old plan or package, if you are replacing it. For more information about this option, see the bind option VALIDATE in Chapter 2 of *DB2 Command Reference*.

3 RESOURCE RELEASE TIME

Lets you specify COMMIT or DEALLOCATE to tell when to release locks on resources. The default is that used for the old plan or package, if you are replacing it. For a description of the effects of those values, see "The ACQUIRE and RELEASE options" on page 337.

4 EXPLAIN PATH SELECTION

Lets you specify YES or NO for whether to obtain EXPLAIN information about how SQL statements in the package execute. The default is NO.

The bind process inserts information into the table *owner*.PLAN_TABLE, where *owner* is the authorization ID of the plan or package owner. If you defined *owner*.DSN_STATEMNT_TABLE, DB2 also inserts information about the cost of statement execution into that table. If you specify YES in this field and BIND in the VALIDATION TIME field, and if you do not correctly define PLAN_TABLE, the bind fails.

For information on EXPLAIN and creating a PLAN_TABLE, see "Obtaining PLAN_TABLE information from EXPLAIN" on page 668.

5 DATA CURRENCY

Lets you specify YES or NO for whether you need data currency for ambiguous cursors opened at remote locations.

Data is current if the data within the host structure is identical to the data within the base table. Data is always current for local processing. For more information on data currency, see "Maintaining data currency" on page 390.

6 PARALLEL DEGREE

Lets you specify ANY to run queries using parallel processing (when possible) or 1 to request that DB2 not execute queries in parallel. See "Chapter 27. Parallel operations and query performance" on page 717 for more information about this option.

8 REOPTIMIZE FOR INPUT VARS

Specifies whether DB2 determines access paths at bind time and again at execution time for statements that contain:
• Input host variables

- Parameter markers
- Special registers

If you specify YES, DB2 determines the access paths again at execution time. When you specify YES for this option, you must also specify YES for DEFER PREPARE, or you will receive a bind error.

9 DEFER PREPARE

Lets you defer preparation of dynamic SQL statements until DB2 encounters the first OPEN, DESCRIBE, or EXECUTE statement that refers to those statements. Specify YES to defer preparation of the statement. For information on using this option, see "Use bind options that improve performance" on page 380.

10 KEEP DYN SQL PAST COMMIT

Specifies whether DB2 keeps dynamic SQL statements after commit points. YES causes DB2 to keep dynamic SQL statements after commit points. An application can execute a PREPARE statement for a dynamic SQL statement once and execute that statement after later commit points without executing PREPARE again. For more information, see "Performance of static and dynamic SQL" on page 497.

11 DBPROTOCOL

Specifies whether DB2 uses DRDA protocol or DB2 private protocol to execute statements that contain 3-part names. For more information, see "Chapter 19. Planning to access distributed data" on page 367.

12 APPLICATION ENCODING

Specifies the application encoding scheme to be used:

blank Indicates that all host variables in static SQL statements are encoded using the encoding scheme in the DEF ENCODING SCHEME field of installation panel DSNTIPF.

ASCII Indicates that the CCSIDs for all host variables in static SQL statements are determined by the values in the ASCII CODED CHAR SET and MIXED DATA fields of installation panel DSNTIPF.

EBCDIC Indicates that the CCSIDs for all host variables in static SQL statements are determined by the values in the EBCDIC CODED CHAR SET and MIXED DATA fields of installation panel DSNTIPF.

UNICODE Indicates that the CCSIDs of all host variables in static SQL statements are determined by the value in the UNICODE CCSID field of installation panel DSNTIPF.

ccsid Specifies a CCSID that determines the set of CCSIDs that are used for all host variables in static SQL statements. If you specify *ccsid*, this value should be a mixed CCSID. For Unicode, the mixed CCSID is a UTF-8 CCSID. DB2 derives the SBCS and DBCS CCSIDs.

13 OPTIMIZATION HINT

Specifies whether you want to use optimization hints to determine access paths. Specify 'hint-id' to indicate that you want DB2 to use the optimization hints in *owner*.PLAN_TABLE, where *owner* is the authorization ID of the plan or package owner. 'hint-id' is a delimited string of up to 8 characters that DB2 compares to the value of OPTHINT in *owner*.PLAN_TABLE to determine the rows to use for optimization hints. If you specify a nonblank

value for 'hint-id', DB2 uses optimization hints only if the value of field OPTIMIZATION HINTS on installation panel DSNTIP4 is YES.

Blank means that you do not want DB2 to use optimization hints. This is the default. For more information, see Part 5 (Volume 2) of *DB2 Administration Guide*.

14 IMMEDIATE WRITE

Specifies when DB2 writes the changes for updated group buffer pool-dependent pages. This field applies only to a data sharing environment. The values that you can specify are:

NO   For transactions that are committed, write the changes at or before phase 2 of the commit process. For transactions that are rolled back, write the changes at the end of the abort process. NO is the default.

YES   Write the changes immediately after group buffer pool-dependent pages are updated.

PH1   Write the changes at or before phase 1 of the commit process. If the transaction is rolled back later, write the additional changes that are caused by the rollback at the end of the abort process.

For more information about this option, see the bind option IMMEDWRITE in Chapter 2 of *DB2 Command Reference*.

15 DYNAMIC RULES

For plans, lets you specify whether run-time (RUN) or bind-time (BIND) rules apply to dynamic SQL statements at run time.

For packages, lets you specify whether run-time (RUN) or bind-time (BIND) rules apply to dynamic SQL statements at run time. For packages that run under an active user-defined function or stored procedure environment, the INVOKEBIND, INVOKERUN, DEFINEBIND, and DEFINERUN options indicate who must have authority to execute dynamic SQL statements in the package.

For packages, the default rules for a package on the local server are the same as the rules for the plan to which the package appends at run time. For a package on the remote server, the default is RUN.

If you specify rules for a package that are different from the rules for the plan, the SQL statements for the package use the rules you specify for that package. If a package that is bound with DEFINEBIND or INVOKEBIND is not executing under an active stored procedure or user-defined function environment, SQL statements for that package use BIND rules. If a package that is bound with DEFINERUN or INVOKERUN is not executing under an active stored procedure or user-defined function environment, SQL statements for that package use RUN rules.

For more information, see "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415.

For packages:

7 SQLERROR PROCESSING

Lets you specify CONTINUE to continue to create a package after finding SQL errors, or NOPACKAGE to avoid creating a package after finding SQL errors.

For plans:

7 RESOURCE ACQUISITION TIME

Lets you specify when to acquire locks on resources. Use:

USE (default) to open table spaces and acquire locks only when the program bound to the plan first uses them.

ALLOCATE to open all table spaces and acquire all locks when you allocate the plan. This value has no effect on dynamic SQL.

For a description of the effects of those values, see "The ACQUIRE and RELEASE options" on page 337.

16 SQLRULES

Lets you specify whether a CONNECT (Type 2) statement executes according to DB2 rules (DB2) or the SQL standard (STD). For information, see "Specifying the SQL rules" on page 418.

17 DISCONNECT

Lets you specify which remote connections end during a commit or a rollback. Regardless of what you specify, all connections in the released-pending state end during commit.

Use:

EXPLICIT to end connections in the release-pending state only at COMMIT or ROLLBACK

AUTOMATIC to end all remote connections

CONDITIONAL to end remote connections that have no open cursors WITH HOLD associated with them.

See the DISCONNECT option of the BIND PLAN subcommand in Chapter 2 of *DB2 Command Reference* for more information about these values.

The System Connection Types panel

This panel displays if you enter YES for ENABLE/DISABLE CONNECTIONS? on the BIND or REBIND PACKAGE or PLAN panels. For BIND or REBIND PACKAGE, the REMOTE option does not display as it does in the following panel.

```
DSNEBP13   SYSTEM CONNECTION TYPES FOR BIND ...            SSID: DSN
 COMMAND ===>

 Select system connection types to be Enabled/Disabled:

 1   ENABLE ALL CONNECTION TYPES? ===>    (* to enable all types)
 or
 2   ENABLE/DISABLE SPECIFIC CONNECTION TYPES ===>    (E/D)

     BATCH ....... ===>    (Y/N)           SPECIFY CONNECTION NAMES?
     DB2CALL ..... ===>    (Y/N)
     RRSAF ....... ===>    (Y/N)
     CICS ........ ===>    (Y/N)           ===> N  (Y/N)
     IMS ......... ===>    (Y/N)
     DLIBATCH .... ===>    (Y/N)           ===> N  (Y/N)
     IMSBMP ...... ===>    (Y/N)           ===> N  (Y/N)
     IMSMPP ...... ===>    (Y/N)           ===> N  (Y/N)
     REMOTE ...... ===>    (Y/N)           ===> N  (Y/N)
```

Figure 135. The system connection types panel

To enable or disable connection types (that is, allow or prevent the connection from running the package or plan), enter the information shown below.

1 ENABLE ALL CONNECTION TYPES?

Lets you enter an asterisk (*) to enable all connections. After that entry, you can ignore the rest of the panel.

2 ENABLE/DISABLE SPECIFIC CONNECTION TYPES

Lets you specify a list of types to enable or disable; you cannot enable some types and disable others in the same operation. If you list types to enable, enter E; that *disables* all other connection types. If you list types to disable, enter D; that *enables* all other connection types. For more information about this option, see the bind options ENABLE and DISABLE in Chapter 2 of *DB2 Command Reference*.

For each connection type that follows, enter Y (yes) if it is on your list, N (no) if it is not. The connection types are:
- **BATCH** for a TSO connection
- **DB2CALL** for a CAF connection
- **RRSAF** for an RRSAF connection
- **CICS** for a CICS connection
- **IMS** for all IMS connections: DLIBATCH, IMSBMP, and IMSMPP
- **DLIBATCH** for a DL/I Batch Support Facility connection
- **IMSBMP** for an IMS connection to a BMP region
- **IMSMPP** for an IMS connection to an MPP or IFP region
- **REMOTE** for remote location names and LU names

For each connection type that has a second arrow, under SPECIFY CONNECTION NAMES?, enter Y if you want to list specific connection names of that type. Leave N (the default) if you do not. If you use Y in any of those fields, you see another panel on which you can enter the connection names. For more information, see "Panels for entering lists of values".

If you use the DISPLAY command under TSO on this panel, you can determine what you have currently defined as "enabled" or "disabled" in your ISPF DSNSPFT library (member DSNCONNS). The information does not reflect the current state of the DB2 Catalog.

If you type DISPLAY ENABLED on the command line, you get the connection names that are currently enabled for your TSO connection types. For example:

```
Display OF ALL   connection name(s) to be  ENABLED

        CONNECTION            SUBSYSTEM
        CICS1                 ENABLED
        CICS2                 ENABLED
        CICS3                 ENABLED
        CICS4                 ENABLED
        DLI1                  ENABLED
        DLI2                  ENABLED
        DLI3                  ENABLED
        DLI4                  ENABLED
        DLI5                  ENABLED
```

Panels for entering lists of values

Some fields in DB2I panels are associated with command keywords that accept multiple values. Those fields lead you to a list panel that lets you enter or modify a an unlimited number of values. A list panel looks like an ISPF edit session and lets you scroll and use a limited set of commands.

The format of each list panel varies, depending on the content and purpose for the panel. Figure 136 is a generic sample of a list panel:

```
panelid            Specific subcommand function              SSID: DSN
 COMMAND ===>_                              SCROLL ===>

 Subcommand operand values:

 CMD
 """"      value  ...
 """"      value  ...
 """"
 """"
 """"
 """"
```

Figure 136. Generic example of a DB2I list panel

For the syntax of specifying names on a list panel, see Chapter 2 of *DB2 Command Reference* for the type of name you need to specify.

All of the list panels let you enter limited commands in two places:
- On the system command line, prefixed by ====>
- In a special command area, identified by """"

On the system command line, you can use:

END   Saves all entered variables, exits the table, and continues to process.

CANCEL
          Discards all entered variables, terminates processing, and returns to the previous panel.

SAVE   Saves all entered variables and remains in the table.

In the special command area, you can use:
Inn   Insert nn lines after this one.
Dnn   Delete this and the following lines for nn lines.
Rnn   Repeat this line nn number of times.

The default for nn is 1.

When you finish with a list panel, specify END to same the current panel values and continue processing.

The Program Preparation: Compile, Link, and Run panel

The second of the program preparation panels (Figure 137) lets you do the last two steps in the program preparation process (compile and link-edit), as well as the PL/I MACRO PHASE for programs requiring this option. For TSO programs, the panel also lets you run programs.

```
DSNEPP02    PROGRAM PREP: COMPILE, PRELINK, LINK, AND RUN    SSID: DSN
 COMMAND ===>_

 Enter compiler or assembler options:
  1  INCLUDE LIBRARY ===> SRCLIB.DATA
  2  INCLUDE LIBRARY ===>
  3  OPTIONS ....... ===> NUM, OPTIMIZE, ADV

 Enter linkage editor options:
  4  INCLUDE LIBRARY ===> SAMPLIB.COBOL
  5  INCLUDE LIBRARY ===>
  6  INCLUDE LIBRARY ===>
  7  LOAD LIBRARY .. ===> RUNLIB.LOAD
  8  PRELINK OPTIONS ===>
  9  LINK OPTIONS... ===>
 Enter run options:
 10  PARAMETERS .... ===> D01, D02, D03/
 11  SYSIN DATA SET  ===> TERM
 12  SYSPRINT DS ... ===> TERM
```

Figure 137. The program preparation: Compile, link, and run panel

1,2 INCLUDE LIBRARY

> Lets you specify up to two libraries containing members for the compiler to include. The members can also be output from DCLGEN. You can leave these fields blank if you wish. There is no default.

3 OPTIONS

> Lets you specify compiler, assembler, or PL/I macro processor options. You can also enter a list of compiler or assembler options by separating entries with commas, blanks, or both. You can leave these fields blank if you wish. There is no default.

4,5,6 INCLUDE LIBRARY

> Lets you enter the names of up to three libraries containing members for the linkage editor to include. You can leave these fields blank if you wish. There is no default.

7 LOAD LIBRARY

> Lets you specify the name of the library to hold the load module. The default value is RUNLIB.LOAD.

> If the load library specified is a PDS, and the input data set is a PDS, the member name specified in INPUT DATA SET NAME field of the Program Preparation panel is the load module name. If the input data set is sequential, the second qualifier of the input data set is the load module name.

> You must fill in this field if you request LINK or RUN on the Program Preparation panel.

8 PRELINK OPTIONS

> Lets you enter a list of prelinker options. Separate items in the list with commas, blanks, or both. You can leave this field blank if you wish. There is no default.

> The prelink utility applies only to programs using C, C++, and IBM COBOL for MVS & VM. See *OS/390 Language Environment for OS/390 & VM Programming Guide* for more information about prelinker options.

9 LINK OPTIONS

Lets you enter a list of link-edit options. Separate items in the list with commas, blanks, or both.

To prepare a program that uses 31-bit addressing and runs above the 16-megabyte line, specify the following link-edit options: AMODE=31, RMODE=ANY.

10 PARAMETERS

Lets you specify a list of parameters you want to pass either to your host language run-time processor, or to your application. Separate items in the list with commas, blanks, or both. You can leave this field blank.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Use a slash (/) to separate the options for your run-time processor from those for your program.

- For PL/I and FORTRAN, run-time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.

  ```
  run-time processor parameters / application parameters
  ```

- For COBOL, reverse this order. Run-time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.

- For assembler and C, there is no supported run-time environment, and you need not use a slash to pass parameters to the application program.

11 SYSIN DATA SET

Lets you specify the name of a SYSIN (or in FORTRAN, FT05F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

12 SYSPRINT DS

Lets you specify the names of a SYSPRINT (or in FORTRAN, FT06F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Your application could need other data sets besides SYSIN and SYSPRINT. If so, remember to catalog and allocate them before you run your program.

When you press ENTER after entering values in this panel, DB2 compiles and link-edits the application. If you specified in the DB2 PROGRAM PREPARATION panel that you want to run the application, DB2 also runs the application.

Chapter 21. Testing an application program

This section discusses how to set up a test environment, test SQL statements, debug your programs, and read output from the precompiler.

Establishing a test environment

This section describes how to design a test data structure and how to fill tables with test data.

> **CICS**
>
> Before you run an application, ensure that the corresponding entries in the RCT, SNT, and RACF control areas authorize your application to run. The system administrator is responsible for these functions; see Part 3 (Volume 1) of *DB2 Administration Guide* for more information on the functions.
>
> In addition, ensure that the transaction code assigned to your program, and to the program itself, is defined to the CICS CSD.

Designing a test data structure

When you test an application that accesses DB2 data, you should have DB2 data available for testing. To do this, you can create test tables and views.

Test Views of Existing Tables. If your application does not change a set of DB2 data and the data exists in one or more production-level tables, you might consider using a view of existing tables.

Test Tables. To create a test table, you need a database and table space. Talk with your DBA to make sure that a database and table spaces are available for your use.

If the data that you want to change already exists in a table, consider using the LIKE clause of CREATE TABLE. If you want others besides yourself to have ownership of a table for test purposes, you can specify a secondary ID as the owner of the table. You can do this with the SET CURRENT SQLID statement; for details, see Chapter 5 of *DB2 SQL Reference*. See Part 3 (Volume 1) of *DB2 Administration Guide* for more information on authorization IDs.

If your location has a separate DB2 system for testing, you can create the test tables and views on the test system, then test your program thoroughly on that system. This chapter assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

Analyzing application data needs

To design test tables and views, first analyze your application's data needs.

1. List the data your application accesses and describe how it accesses each data item. For example, suppose you are testing an application that accesses the DSN8710.EMP, DSN8710.DEPT, and DSN8710.PROJ tables. You might record the information about the data as shown in Table 55.

Table 55. Description of the application's data

Table or View Name	Insert Rows?	Delete Rows?	Column Name	Data Type	Update Access?
DSN8710.EMP	No	No	EMPNO	CHAR(6)	
			LASTNAME	VARCHAR(15)	
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	Yes
			JOB	DECIMAL(3)	Yes
DSN8710.DEPT	No	No	DEPTNO	CHAR(3)	
			MGRNO	CHAR (6)	
DSN8710.PROJ	Yes	Yes	PROJNO	CHAR(6)	
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	Yes

2. Determine the test tables and views you need to test your application.

   Create a test table on your list when either:
   - The application modifies data in the table
   - You need to create a view based on a test table because your application modifies the view's data.

   To continue the example, create these test tables:
   - TEST.EMP, with the following format:

EMPNO	LASTNAME	WORKDEPT	PHONENO	JOB
:	:	:	:	:
:	:	:	:	:

   - TEST.PROJ. with the same columns and format as DSN8710.PROJ, because the application inserts rows into the DSN8710.PROJ table.

   To support the example, create a test view of the DSN8710.DEPT table. Because the application does not change any data in the DSN8710.DEPT table, you can base the view on the table itself (rather than on a test table). However, it is safer to have a complete set of test tables and to test the program thoroughly using only test data. The TEST.DEPT view has the following format:

DEPTNO	MGRNO
:	:
:	:

Obtaining authorization

Before you can create a table, you need to be authorized to create tables and to use the table space in which the table is to reside. You must also have authority to bind and run programs you want to test. Your DBA can grant you the authorization needed to create and access tables and to bind and run programs.

If you intend to use existing tables and views (either directly or as the basis for a view), you need privileges to access those tables and views. Your DBA can grant those privileges.

To create a view, you must have authorization for each table and view on which you base the view. You then have the same privileges over the view that you have over the tables and views on which you based the view. Before trying the examples, have your DBA grant you the privileges to create new tables and views and to access existing tables. Obtain the names of tables and views you are authorized to access (as well as the privileges you have for each table) from your DBA. See "Chapter 2. Working with tables and modifying data" on page 17 for more information on creating tables and views.

Creating a comprehensive test structure

The following SQL statements shows how to create a complete test structure to contain a small table named SPUFINUM. The test structure consists of:
- A storage group named SPUFISG
- A database named SPUFIDB
- A table space named SPUFITS in SPUFIDB and using SPUFISG
- A table named SPUFINUM within the table space SPUFITS

```
CREATE STOGROUP SPUFISG
   VOLUMES (user-volume-number)
   VCAT DSNCAT ;

CREATE DATABASE SPUFIDB ;

CREATE TABLESPACE SPUFITS
   IN SPUFIDB
   USING STOGROUP SPUFISG ;

CREATE TABLE SPUFINUM
   ( XVAL CHAR(12) NOT NULL,
     ISFLOAT FLOAT,
     DEC30 DECIMAL(3,0),
     DEC31 DECIMAL(3,1),
     DEC32 DECIMAL(3,2),
     DEC33 DECIMAL(3,3),
     DEC10 DECIMAL(1,0),
     DEC11 DECIMAL(1,1),
     DEC150 DECIMAL(15,0),
     DEC151 DECIMAL(15,1),
     DEC1515 DECIMAL(15,15) )
   IN SPUFIDB.SPUFITS ;
```

For details about each CREATE statement, see *DB2 SQL Reference*.

Filling the tables with test data

There are several ways in which you can put test data into a table:
- INSERT ... VALUES (an SQL statement) puts one row into a table each time the statement executes. For information on the INSERT statement, see "Inserting a row: INSERT" on page 25.
- INSERT ... SELECT (an SQL statement) obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement. For information on this technique, see "Filling a table from another table: Mass INSERT" on page 26.
- The LOAD utility obtains data from a sequential file (a non-DB2 file), formats it for a table, and puts it into a table. For more details about the LOAD utility, see *DB2 Utility Guide and Reference*.
- The DB2 sample UNLOAD program (DSNTIAUL) can unload data from a table or view and build control statements for the LOAD utility. See "Appendix C. How to run sample programs DSNTIAUL, DSNTIAD, and DSNTEP2" on page 835 for more information about the sample UNLOAD program.

| • The UNLOAD utility can unload data from a table and build control statements for
| the LOAD utility. See Part 2 of *DB2 Utility Guide and Reference* for more
| information about the UNLOAD utility.

Testing SQL statements using SPUFI

You can use SPUFI (an interface between ISPF and DB2) to test SQL statements
in a TSO/ISPF environment. With SPUFI panels you can put SQL statements into a
data set that DB2 subsequently executes. The SPUFI Main panel has several
functions that permit you to:

• Name an input data set to hold the SQL statements passed to DB2 for execution
• Name an output data set to contain the results of executing the SQL statements
• Specify SPUFI processing options.

SQL statements executed under SPUFI operate on actual tables (in this case, the
tables you have created for testing). Consequently, before you access DB2 data:

• Make sure that all tables and views your SQL statements refer to exist
• If the tables or views do not exist, create them (or have your database
  administrator create them). You can use SPUFI to issue the CREATE statements
  used to create the tables and views you need for testing.

For more details about how to use SPUFI, see "Chapter 5. Executing SQL from
your terminal using SPUFI" on page 51.

Debugging your program

Many sites have guidelines regarding what to do if your program abends. The
following suggestions are some common ones.

Debugging programs in TSO

Documenting the errors returned from test helps you investigate and correct
problems in the program. The following information can be useful:

• The application plan name of the program
• The input data being processed
• The failing SQL statement and its function
• The contents of the SQLCA (SQL communication area) and, if your program
  accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
• The date and time of day
• The abend code and any error messages.

When your program encounters an error that does not result in an abend, it can
pass all the required error information to a standard error routine. Online programs
might also send an error message to the terminal.

Language test facilities

For information on the compiler or assembler test facilities see the publications for
the compiler or CODE/370. The compiler publications include information on the
appropriate debugger for the language you are using.

The TSO TEST command

The TSO TEST command is especially useful for debugging assembler programs.

The following example is a command procedure (CLIST) that runs a DB2 application named MYPROG under TSO TEST, and sets an address stop at the entry to the program. The DB2 subsystem name in this example is DB4.

```
PROC 0
TEST 'prefix.SDSNLOAD(DSN)' CP
DSN SYSTEM(DB4)
AT MYPROG.MYPROG.+0 DEFER
GO
RUN  PROGRAM(MYPROG)  LIBRARY('L186331.RUNLIB.LOAD(MYPROG)')
```

For more information about the TEST command, see *OS/390 TSO/E Command Reference*.

ISPF Dialog Test is another option to help you in the task of debugging.

Debugging programs in IMS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input message being processed
- The name of the originating logical terminal
- The failing statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The program's PSB name
- The transaction code that the program was processing
- The call function (that is, the name of a DL/I function)
- The contents of the PCB that the program's call refers to
- If a DL/I database call was running, the SSAs, if any, that the call used
- The abend completion code, abend reason code, and any dump error messages.

When your program encounters an error, it can pass all the required error information to a standard error routine. Online programs can also send an error message to the originating logical terminal.

An interactive program also can send a message to the master terminal operator giving information about the program's termination. To do that, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls.

Some sites run a BMP at the end of the day to list all the errors that occurred during the day. If your location does this, you can send a message using an express PCB that has its destination set for that BMP.

Batch Terminal Simulator (BTS): The Batch Terminal Simulator (BTS) allows you to test IMS application programs. BTS traces application program DL/I calls and SQL statements, and simulates data communication functions. It can make a TSO terminal appear as an IMS terminal to the terminal operator, allowing the end user to interact with the application as though it were online. The user can use any application program under the user's control to access any database (whether DL/I or DB2) under the user's control. Access to DB2 databases requires BTS to operate

in batch BMP or TSO BMP mode. For more information on the Batch Terminal Simulator, see *IMS Batch Terminal Simulator General Information*.

Debugging programs in CICS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input data being processed
- The ID of the originating logical terminal
- The failing SQL statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- Data peculiar to CICS that you should record
- Abend code and dump error messages
- Transaction dump, if produced.

Using CICS facilities, you can have a printed error record; you can also print the SQLCA (and SQLDA) contents.

Debugging aids for CICS

CICS provides the following aids to the testing, monitoring, and debugging of application programs:

Execution (Command Level) Diagnostic Facility (EDF). EDF shows CICS commands for all releases of CICS. See "CICS execution diagnostic facility" for more information. If you are using an earlier version of CICS, the CALL TO RESOURCE MANAGER DSNCSQL screen displays a status of ″ABOUT TO EXECUTE″ or ″COMMAND EXECUTION COMPLETE.″

Abend recovery. You can use the HANDLE ABEND command to deal with abend conditions, and the ABEND command to cause a task to abend.

Trace facility. A trace table can contain entries showing the execution of various CICS commands, SQL statements, and entries generated by application programs; you can have it written to main storage and, optionally, to an auxiliary storage device.

Dump facility. You can specified areas of main storage to dump onto a sequential data set, either tape or disk, for subsequent offline formatting and printing with a CICS utility program.

Journals. For statistical or monitoring purposes, facilities can create entries in special data sets called journals. The system log is a journal.

Recovery. When an abend occurs, CICS restores certain resources to their original state so that the operator can easily resubmit a transaction for restart. You can use the SYNCPOINT command to subdivide a program so that you only need to resubmit the uncompleted part of a transaction.

For more details about each of these topics, see *CICS for MVS/ESA Application Programming Reference*.

CICS execution diagnostic facility

The CICS execution diagnostic facility (EDF) traces SQL statements in an interactive debugging mode, enabling application programmers to test and debug programs online without changing the program or the program preparation procedure.

EDF intercepts the running application program at various points and displays helpful information about the statement type, input and output variables, and any error conditions after the statement executes. It also displays any screens that the application program sends, making it possible to converse with the application program during testing just as a user would on a production system.

EDF displays essential information before and after an SQL statement, while the task is in EDF mode. This can be a significant aid in debugging CICS transaction programs containing SQL statements. The SQL information that EDF displays is helpful for debugging programs and for error analysis after an SQL error or warning. Using this facility reduces the amount of work you need to do to write special error handlers.

EDF before execution: Figure 138 is an example of an EDF screen before it executes an SQL statement. The names of the key information fields on this panel are in **boldface**.

```
TRANSACTION: XC05   PROGRAM: TESTC05   TASK NUMBER: 0000668   DISPLAY:  00
STATUS:  ABOUT TO EXECUTE COMMAND
CALL TO RESOURCE MANAGER DSNCSQL
EXEC SQL INSERT
 DBRM=TESTC05,  STMT=00368, SECT=00004
 IVAR 001: TYPE=CHAR,              LEN=00007,   IND=000    AT X'03C92810'
           DATA=X'F0F0F9F4F3F4F2'
 IVAR 002: TYPE=CHAR,              LEN=00007,   IND=000    AT X'03C92817'
           DATA=X'F0F1F3F3F7F5F1'
 IVAR 003: TYPE=CHAR,              LEN=00004,   IND=000    AT X'03C9281E'
           DATA=X'E7C3F0F5'
 IVAR 004: TYPE=CHAR,              LEN=00040,   IND=000    AT X'03C92822'
           DATA=X'E3C5E2E3C3F0F540E2C9D4D7D3C540C4C2F240C9D5E2C5D9E3404040'...
 IVAR 005: TYPE=SMALLINT,          LEN=00002,   IND=000    AT X'03C9284A'
           DATA=X'0001'

 OFFSET:X'001ECE'   LINE:UNKNOWN        EIBFN=X'1002'


ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 138. EDF screen before a DB2 SQL statement

The DB2 SQL information in this screen is as follows:

- EXEC SQL *statement type*

  This is the type of SQL statement to execute. The SQL statement can be any valid SQL statement, such as COMMIT, DROP TABLE, EXPLAIN, FETCH, or OPEN.

- DBRM=*dbrm name*

  The name of the database request module (DBRM) currently processing. The DBRM, created by the DB2 precompiler, contains information about an SQL statement.

- STMT=*statement number*

  This is the DB2 precompiler-generated statement number. The source and error message listings from the precompiler use this statement number, and you can

use it to determine which statement is processing. This number is a source line counter that includes host language statements. A statement number greater than 32,767 displays as 0.

- SECT=*section number*

  The section number of the plan that the SQL statement uses.

SQL statements containing input host variables: The IVAR (input host variables) section and its attendant fields only appear when the executing statement contains input host variables.

The host variables section includes the variables from predicates, the values used for inserting or updating, and the text of dynamic SQL statements being prepared. The address of the input variable is AT '*nnnnnnnn*'.

Additional host variable information:

- TYPE=*data type*

  Specifies the data type for this host variable. The basic data types include character string, graphic string, binary integer, floating-point, decimal, date, time, and timestamp. For additional information refer to "Data types" on page 3.

- LEN=*length*

  Length of the host variable.

- IND=*indicator variable status number*

  Represents the indicator variable associated with this particular host variable. A value of zero indicates that no indicator variable exists. If the value for the selected column is null, DB2 puts a negative value in the indicator variable for this host variable. For additional information refer to "Using indicator variables with host variables" on page 70.

- DATA=*host variable data*

  The data, displayed in hexadecimal format, associated with this host variable. If the data exceeds what can display on a single line, three periods (...) appear at the far right to indicate more data is present.

EDF after execution: Figure 139 on page 469 shows an example of the first EDF screen displayed after the executing an SQL statement. The names of the key information fields on this panel are in **boldface**.

```
  TRANSACTION: XC05  PROGRAM: TESTC05    TASK NUMBER: 0000698   DISPLAY:  00
  STATUS: COMMAND EXECUTION COMPLETE
  CALL TO RESOURCE MANAGER DSNCSQL
  EXEC SQL FETCH                        P.AUTH=SYSADM  , S.AUTH=
  PLAN=TESTC05, DBRM=TESTC05,  STMT=00346, SECT=00001
  SQL COMMUNICATION AREA:
   SQLCABC    = 136                                          AT X'03C92789'
   SQLCODE    = 000                                          AT X'03C9278D'
   SQLERRML   = 000                                          AT X'03C92791'
   SQLERRMC   = ''                                           AT X'03C92793'
   SQLERRP    = 'DSN'                                        AT X'03C927D9'
   SQLERRD(1-6) = 000,  000, 00000, -1, 00000,  000         AT X'03C927E1'
   SQLWARN(0-A) = '_ _ _ _ _ _ _ _ _ _ _'                   AT X'03C927F9'
   SQLSTATE   = 00000                                        AT X'03C92804'
+ OVAR 001: TYPE=INTEGER,             LEN=00004,    IND=000  AT X'03C920A0'
          DATA=X'00000001'
  OFFSET:X'001D14'  LINE:UNKNOWN        EIBFN=X'1802'



ENTER:   CONTINUE
PF1  : UNDEFINED         PF2 : UNDEFINED          PF3 : END EDF SESSION
PF4  : SUPPRESS DISPLAYS PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7  : SCROLL BACK       PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 139. EDF screen after a DB2 SQL statement

The DB2 SQL information in this screen is as follows:

- P.AUTH=*primary authorization ID*

  The primary DB2 authorization ID.

- S.AUTH=*secondary authorization ID*

  If the RACF list of group options is not active, then DB2 uses the connected group name that the CICS attachment facility supplies as the secondary authorization ID. If the RACF list of group options is active, then DB2 ignores the connected group name that the CICS attachment facility supplies, but the value appears in the DB2 list of secondary authorization IDs.

- PLAN=*plan name*

  The name of plan that is currently running. The PLAN represents the control structure produced during the bind process and used by DB2 to process SQL statements encountered while the application is running.

- SQL Communication Area (SQLCA)

  The SQLCA contains information about errors, if any occur. After returning from DB2, the information is available. DB2 uses the SQLCA to give an application program information about the executing SQL statements.

Plus signs (+) on the left of the screen indicate that you can see additional EDF output by using PF keys to scroll the screen forward or back.

The OVAR (output host variables) section and its attendant fields only appear when the executing statement returns output host variables.

Figure 140 on page 470 contains the rest of the EDF output for our example.

```
TRANSACTION: XC05   PROGRAM: TESTC05    TASK NUMBER: 0000698   DISPLAY:  00
 STATUS:  COMMAND EXECUTION COMPLETE
 CALL TO RESOURCE MANAGER DSNCSQL
+ OVAR 002: TYPE=CHAR,                 LEN=00008,    IND=000     AT X'03C920B0'
            DATA=X'C8F3E3E3C1C2D3C5'
  OVAR 003: TYPE=CHAR,                 LEN=00040,    IND=000     AT X'03C920B8'
            DATA=X'C9D5C9E3C9C1D340D3D6C1C44040404040404040404040404040404040'...




 OFFSET:X'001D14'    LINE:UNKNOWN        EIBFN=X'1802'


 ENTER:  CONTINUE
 PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : END EDF SESSION
 PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
 PF7 : SCROLL BACK        PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
 PF10: PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 140. EDF screen after a DB2 SQL statement, continued

The attachment facility automatically displays SQL information while in the EDF
mode. (You can start EDF as outlined in the appropriate CICS application
programmer's reference manual.) If this is not the case, contact your installer and
see Part 2 of *DB2 Installation Guide*.

Locating the problem

If your program does not run correctly, you need to isolate the problem. If the DB2
did not invalidate the program's application plan, you should check the following
items:

- Output from the precompiler which consists of errors and warnings. Ensure that
  you have resolved all errors and warnings.
- Output from the compiler or assembler. Ensure that you have resolved all error
  messages.
- Output from the linkage editor.
  - Have you resolved all external references?
  - Have you included all necessary modules in the correct order?
  - Did you include the correct language interface module? The correct language
    interface module is:
    - DSNELI for TSO
    - DFSLI000 for IMS
    - DSNCLI for CICS
    - DSNALI for the call attachment facility.
  - Did you specify the correct entry point to your program?
- Output from the bind process.
  - Have you resolved all error messages?
  - Did you specify a plan name? If not, the bind process assumes you want to
    process the DBRM for diagnostic purposes, but do not want to produce an
    application plan.

– Have you specified all the DBRMs and packages associated with the programs that make up the application and their partitioned data set (PDS) names in a single application plan?
- Your JCL.

> **IMS**
>
> – If you are using IMS, have you included the DL/I option statement in the correct format?

– Have you included the region size parameter in the EXEC statement? Does it specify a region size large enough for the storage required for the DB2 interface, the TSO, IMS, or CICS system, and your program?
– Have you included the names of all data sets (DB2 and non-DB2) that the program requires?
- Your program.

  You can also use dumps to help localize problems in your program. For example, one of the more common error situations occurs when your program is running and you receive a message that it abended. In this instance, your test procedure might be to capture a TSO dump. To do so, you must allocate a SYSUDUMP or SYSABEND dump data set before calling DB2. When you press the ENTER key (after the error message and READY message), the system requests a dump. You then need to FREE the dump data set.

Analyzing error and warning messages from the precompiler

Under some circumstances, the statements that the DB2 precompiler generates can produce compiler or assembly error messages. You must know why the messages occur when you compile DB2-produced source statements. For more information about warning messages, see the following host language sections:
- "Coding SQL statements in an assembler application" on page 107
- "Coding SQL statements in a C or a C++ application" on page 121
- "Coding SQL statements in a COBOL application" on page 141
- "Coding SQL statements in a FORTRAN application" on page 164
- "Coding SQL statements in a PL/I application" on page 174.

SYSTERM output from the precompiler

The DB2 precompiler provides SYSTERM output when you allocate the ddname SYSTERM. If you use the Program Preparation panels to prepare and run your program, DB2I allocates SYSTERM according to the TERM option you specify.

The SYSTERM output provides a brief summary of the results from the precompiler, all error messages that the precompiler generated, and the statement in error, when possible. Sometimes, the error messages by themselves are not enough. In such cases, you can use the line number provided in each error message to locate the failing source statement.

Figure 141 on page 472 shows the format of SYSTERM output.

```
DB2 SQL PRECOMPILER        MESSAGES

DSNH104I E    DSNHPARS LINE 32 COL 26  ILLEGAL SYMBOL "X"  VALID SYMBOLS ARE:, FROM[1]
SELECT VALUE INTO HIPPO X;[2]

DB2 SQL PRECOMPILER        STATISTICS
SOURCE STATISTICS[3]
  SOURCE LINES READ: 36
  NUMBER OF SYMBOLS: 15
  SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 1848
THERE WERE 1 MESSAGES FOR THIS PROGRAM.[4]
THERE WERE 0 MESSAGES SUPPRESSED BY THE FLAG OPTION.[5]
111664 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.[6]
RETURN CODE IS 8[7]
```

Figure 141. DB2 precompiler SYSTERM output

Notes for Figure 141:

1. Error message.
2. Source SQL statement.
3. Summary statements of source statistics.
4. Summary statement of the number of errors detected.
5. Summary statement indicating the number of errors detected but not printed. That value might occur if you specify a FLAG option other than I.
6. Storage requirement statement telling you how many bytes of working storage that the DB2 precompiler actually used to process your source statements. That value helps you determine the storage allocation requirements for your program.
7. Return code: 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.

SYSPRINT output from the precompiler

SYSPRINT output is what the DB2 precompiler provides when you use a procedure to precompile your program. See Table 53 on page 426 for a list of JCL procedures that DB2 provides.

When you use the Program Preparation panels to prepare and run your program, DB2 allocates SYSPRINT according to TERM option you specify (on line 12 of the PROGRAM PREPARATION: COMPILE, PRELINK, LINK, AND RUN panel). As an alternative, when you use the DSNH command procedure (CLIST), you can specify PRINT(TERM) to obtain SYSPRINT output at your terminal, or you can specify PRINT(qualifier) to place the SYSPRINT output into a data set named *authorizationid.qualifier.PCLIST*. Assuming that you do not specify PRINT as LEAVE, NONE, or TERM, DB2 issues a message when the precompiler finishes, telling you where to find your precompiler listings. This helps you locate your diagnostics quickly and easily.

The SYSPRINT output can provide information about your precompiled source module if you specify the options SOURCE and XREF when you start the DB2 precompiler.

The format of SYSPRINT output is as follows:

- A list of the DB2 precompiler options (Figure 142) in effect during the precompilation (if you did not specify NOOPTIONS).

```
DB2 SQL PRECOMPILER        Version 7

OPTIONS SPECIFIED: HOST(PLI),XREF,SOURCE¹

OPTIONS USED - SPECIFIED OR DEFAULTED²
      APOST
      APOSTSQL
      CONNECT(2)
      DEC(15)
      FLAG(I)
   NOGRAPHIC
      HOST(PLI)
      NOT KATAKANA
      LINECOUNT(60)
      MARGINS(2,72)
      ONEPASS
      OPTIONS
      PERIOD
      SOURCE
      STDSQL(NO)
      SQL(DB2)
      XREF
```

Figure 142. DB2 precompiler SYSPRINT output: Options section

Notes for Figure 142:

1. This section lists the options specified at precompilation time. This list does not appear if one of the precompiler option is NOOPTIONS.

2. This section lists the options that are in effect, including defaults, forced values, and options you specified. The DB2 precompiler overrides or ignores any options you specify that are inappropriate for the host language.

- A listing (Figure 143) of your source statements (only if you specified the SOURCE option).

```
DB2 SQL PRECOMPILER        TMN5P40:PROCEDURE OPTIONS (MAIN):        PAGE 2

     1        TMN5P40:PROCEDURE OPTIONS(MAIN) ;                    00000100
     2        /****************************************************00000200
     3        *       program description and prologue                  00000300
   ⋮
  1324        /**********************************************/      00132400
  1325        /*  GET INFORMATION ABOUT THE PROJECT FROM THE  */    00132500
  1326        /*   PROJECT TABLE.                          */       00132600
  1327        /**********************************************/      00132700
  1328             EXEC SQL SELECT ACTNO, PREQPROJ, PREQACT          00132800
  1329                     INTO PROJ_DATA                            00132900
  1330                     FROM TPREREQ                              00133000
  1331                         WHERE PROJNO = :PROJ_NO;              00133100
  1332                                                               00133200
  1333        /**********************************************/      00133300
  1334        /*   PROJECT IS FINISHED. DELETE IT.         */       00133400
  1335        /**********************************************/      00133500
  1336                                                               00133600
  1337             EXEC SQL DELETE FROM PROJ                         00133700
  1338                 WHERE PROJNO = :PROJ_NO;                      00133800
   ⋮
  1523        END;                                                  00152300
```

Figure 143. DB2 precompiler SYSPRINT output: Source statements section

Notes for Figure 143:

– The left column of sequence numbers, which the DB2 precompiler generates, is for use with the symbol cross-reference listing, the precompiler error messages, and the BIND error messages.

– The right column of sequence numbers come from the sequence numbers supplied with your source statements.

• A list (Figure 144) of the symbolic names used in SQL statements (this listing appears only if you specify the XREF option).

```
DB2 SQL PRECOMPILER     SYMBOL CROSS-REFERENCE LISTING          PAGE 29

DATA NAMES                   DEFN     REFERENCE

"ACTNO"                      ****     FIELD
                                      1328
"PREQACT"                    ****     FIELD
                                      1328
"PREQPROJ"                   ****     FIELD
                                      1328
"PROJNO"                     ****     FIELD
                                      1331 1338


 ...


PROJ_DATA                    495      CHARACTER(35)
                                      1329
PROJ_NO                      496      CHARACTER(3)
                                      1331 1338
"TPREREQ"                    ****     TABLE
                                      1330 1337
```

Figure 144. DB2 precompiler SYSPRINT output: Symbol cross-reference section

Notes for Figure 144:

DATA NAMES
Identifies the symbolic names used in source statements. Names enclosed in quotation marks (") or apostrophes (') are names of SQL entities such as tables, columns, and authorization IDs. Other names are host variables.

DEFN
Is the number of the line that the precompiler generates to define the name. **** means that the object was not defined or the precompiler did not recognize the declarations.

REFERENCE
Contains two kinds of information: what the source program defines the symbolic name to be, and which lines refer to the symbolic name. If the symbolic name refers to a valid host variable, the list also identifies the data type or STRUCTURE.

• A summary (Figure 145 on page 475) of the errors detected by the DB2 precompiler and a list of the error messages generated by the precompiler.

```
DB2 SQL PRECOMPILER          STATISTICS

  SOURCE STATISTICS
    SOURCE LINES READ: 1523[1]
    NUMBER OF SYMBOLS: 128[2]
    SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 6432[3]

  THERE WERE 1 MESSAGES FOR THIS PROGRAM.[4]
  THERE WERE 0 MESSAGES SUPPRESSED.[5]
  65536 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.[6]
  RETURN CODE IS 8.[7]
  DSNH104I E LINE 590 COL 64 ILLEGAL SYMBOL: 'X'; VALID SYMBOLS ARE:,FROM[8]
```

Figure 145. DB2 precompiler SYSPRINT output: Summary section

Notes for Figure 145:

1. Summary statement indicating the number of source lines.
2. Summary statement indicating the number of symbolic names in the symbol table (SQL names and host names).
3. Storage requirement statement indicating the number of bytes for the symbol table.
4. Summary statement indicating the number of messages printed.
5. Summary statement indicating the number of errors detected but not printed. You might get this statement if you specify the option FLAG.
6. Storage requirement statement indicating the number of bytes of working storage actually used by the DB2 precompiler to process your source statements.
7. Return code—0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.
8. Error messages (this example detects only one error).

Chapter 22. Processing DL/I batch applications

This chapter describes DB2 support for DL/I batch applications under these headings:
- "Planning to use DL/I batch"
- "Program design considerations" on page 478
- "Input and output data sets" on page 480
- "Program preparation considerations" on page 482
- "Restart and recovery" on page 484

Planning to use DL/I batch

Features and functions of DB2 DL/I batch support, below, tells what you can do in a DL/I batch program. "Requirements for using DB2 in a DL/I batch job" on page 478 tells, in general, what you must do to make it happen.

Features and functions of DB2 DL/I batch support

A batch DL/I program can issue:
- Any IMS batch call, except ROLS, SETS, and SYNC calls. ROLS and SETS calls provide intermediate backout point processing, which DB2 does not support. The SYNC call provides commit point processing without identifying the commit point with a value. IMS does not allow a SYNC call in batch, and neither does the DB2 DL/I batch support.

  Issuing a ROLS, SETS or SYNC call in an application program causes a system abend X'04E' with the reason code X'00D44057' in register 15.
- GSAM calls.
- IMS system services calls.
- Any SQL statements, except COMMIT and ROLLBACK. IMS and CICS environments do not allow those SQL statements. The application program must use the IMS CHKP call to commit data and the IMS ROLL or ROLB to roll back changes.

  Issuing a COMMIT statement causes SQLCODE -925; issuing a ROLLBACK statement causes SQLCODE -926. Those statements also return SQLSTATE '2D521'.
- Any call to a standard or traditional access method (for example, QSAM, VSAM, and so on).

The restart capabilities for DB2 and IMS databases, as well as for sequential data sets accessed through GSAM, are available through the IMS Checkpoint and Restart facility.

DB2 allows access to both DB2 and DL/I data through the use of the following DB2 and IMS facilities:
- IMS synchronization calls, which commit and abend units of recovery
- The DB2 IMS attachment facility, which handles the two-phase commit protocol and allows both systems to synchronize a unit of recovery during a restart after a failure
- The IMS log, used to record the instant of commit.

In a data sharing environment, DL/I batch supports group attachment. You can specify a group attachment name instead of a subsystem name in the SSN

parameter of the DDITV02 data set for the DL/I batch job. See "DB2 DL/I Batch Input" on page 480 for information on the SSN parameter and the DDITV02 data set.

Requirements for using DB2 in a DL/I batch job

Using DB2 in a DL/I batch job requires the following changes to the application program and the job step JCL:

- You must add SQL statements to your application program to gain access to DB2 data. You must then precompile the application program and bind the resulting DBRM into a plan or package, as described in "Chapter 20. Preparing an application program to run" on page 395.

- Before you run the application program, use JOBLIB, STEPLIB, or link book to access the DB2 load library, so that DB2 modules can be loaded.

- In a data set that is specified by a DDITV02 DD statement, specify the program name and plan name for the application, and the connection name for the DL/I batch job.

  In an input data set or in a subsystem member, specify information about the connection between DB2 and IMS. The input data set name is specified with a DDITV02 DD statement. The subsystem member name is specified by the parameter SSM= on the DL/I batch invocation procedure. For detailed information about the contents of the subsystem member and the DDITV02 data set, see "DB2 DL/I Batch Input" on page 480.

- Optionally specify an output data set using the DDOTV02 DD statement. You might need this data set to receive messages from the IMS attachment facility about indoubt and diagnostic information.

Authorization

When the batch application tries to run the first SQL statement, DB2 checks whether the authorization ID has the EXECUTE privilege for the plan. DB2 uses the same ID for later authorization checks and also identifies records from the accounting and performance traces.

The primary authorization ID is the value of the USER parameter on the job statement, if that is available. It is the TSO logon name if the job is submitted. Otherwise, it is the IMS PSB name. In that case, however, the ID must not begin with the string "SYSADM"—which causes the job to abend. The batch job is rejected if you try to change the authorization ID in an exit routine.

Program design considerations

Using DL/I batch can affect your application design and programming in the areas described below.

Address spaces

A DL/I batch region is independent of both the IMS control region and the CICS address space. The DL/I batch region loads the DL/I code into the application region along with the application program.

Commits

Commit IMS batch applications frequently so that you do not tie up resources for an extended time. If you need coordinated commits for recovery, see Part 4 (Volume 1) of *DB2 Administration Guide*.

SQL statements and IMS calls

You cannot use the SQL COMMIT and ROLLBACK statements, which return an SQL error code. You also cannot use ROLS, SETS, and SYNC calls, which cause the application program to abend.

Checkpoint calls

Write your program with SQL statements and DL/I calls, and use checkpoint calls. All checkpoints issued by a batch application program must be unique. The frequency of checkpoints depends on the application design. At a checkpoint, DL/I positioning is lost, DB2 cursors are closed (with the possible exception of cursors defined as WITH HOLD), commit duration locks are freed (again with some exceptions), and database changes are considered permanent to both IMS and DB2.

Application program synchronization

It is possible to design an application program without using IMS checkpoints. In that case, if the program abends before completing, DB2 backs out any updates, and you can use the IMS batch backout utility to back out the DL/I changes.

It is also possible to have IMS dynamically back out the updates within the same job. You must specify the BKO parameter as 'Y' and allocate the IMS log to DASD.

You could have a problem if the system fails after the program terminates, but before the job step ends. If you do not have a checkpoint call before the program ends, DB2 commits the unit of work without involving IMS. If the system fails before DL/I commits the data, then the DB2 data is out of synchronization with the DL/I changes. If the system fails during DB2 commit processing, the DB2 data could be indoubt.

It is recommended that you always issue a symbolic checkpoint at the end of any update job to coordinate the commit of the outstanding unit of work for IMS and DB2. When you restart the application program, you must use the XRST call to obtain checkpoint information and resolve any DB2 indoubt work units.

Checkpoint and XRST considerations

If you use an XRST call, DB2 assumes that any checkpoint issued is a symbolic checkpoint. The options of the symbolic checkpoint call differ from the options of a basic checkpoint call. Using the incorrect form of the checkpoint call can cause problems.

If you do not use an XRST call, then DB2 assumes that any checkpoint call issued is a basic checkpoint.

Checkpoint IDs must be EBCDIC characters to make restart easier.

When an application program needs to be restartable, you must use symbolic checkpoint and XRST calls. If you use an XRST call, it must be the first IMS call issued and must occur before any SQL statement. Also, you must use only one XRST call.

Synchronization call abends

If the application program contains an incorrect IMS synchronization call (CHKP, ROLB, ROLL, or XRST), causing IMS to issue a bad status code in the PCB, DB2 abends the application program. Be sure to test these calls before placing the programs in production.

Input and output data sets

Two data sets need your attention:
- DDITV02 for input
- DDOTV02 for output.

DB2 DL/I Batch Input

Before you can run a DL/I batch job, you need to provide values for a number of input parameters. The input parameters are positional and delimited by commas.

You can specify values for the following parameters using a DDITV02 data set *or* a subsystem member:

```
SSN,LIT,ESMT,RTT,REO,CRC
```

You can specify values for the following parameters *only* in a DDITV02 data set:

```
CONNECTION_NAME,PLAN,PROG
```

If you use the DDITV02 data set and specify a subsystem member, the values in the DDITV02 DD statement override the values in the specified subsystem member. If you provide neither, DB2 abends the application program with system abend code X'04E' and a unique reason code in register 15.

DDITV02 is the DD name for a data set that has DCB options of LRECL=80 and RECFM=F or FB.

A subsystem member is a member in the IMS procedure library. Its name is derived by concatenating the value of the SSM parameter to the value of the IMSID parameter. You specify the SSM parameter and the IMSID parameter when you invoke the DLIBATCH procedure, which starts the DL/I batch processing environment.

The meanings of the input parameters are:

Field **Content**

SSN The name of the DB2 subsystem is required. You must specify a name in order to make a connection to DB2.

The SSN value can be from one to four characters long.

If the value in the SSN parameter is the name of a subsystem in the data sharing group, and that subsystem is active, the application attaches to that subsystem. If the value in the SSN parameter is not the name of an active subsystem, but is a group attachment name, the application attaches to an active DB2 subsystem in the data sharing group. See Chapter 2 of *DB2 Data Sharing: Planning and Administration* for more information about group attachment.

LIT DB2 requires a language interface token to route SQL statements when operating in the online IMS environment. Because a batch application

program can only connect to one DB2 system, DB2 does not use the LIT value. It is recommended that you specify the value as SYS1; however, you can omit it (enter SSN,,ESMT).

The LIT value can be from zero to four characters long.

ESMT The name of the DB2 initialization module, DSNMIN10, is required.

The ESMT value must be eight characters long.

RTT Specifying the resource translation table is optional.

The RTT can be from zero to eight characters long.

REO The region error option determines what to do if DB2 is not operational or the plan is not available. There are three options:

- *R*, the default, results in returning an SQL return code to the application program. The most common SQLCODE issued in this case is -923 (SQLSTATE '57015').
- *Q* results in an abend in the batch environment; however, in the online environment, it places the input message in the queue again.
- *A* results in an abend in both the batch environment and the online environment.

If the application program uses the XRST call, and if coordinated recovery is required on the XRST call, then REO is ignored. In that case, the application program terminates abnormally if DB2 is not operational.

The REO value can be from zero to one character long.

CRC Because DB2 commands are not supported in the DL/I batch environment, the command recognition character is not used at this time.

The CRC value can be from zero to one character long.

CONNECTION_NAME

The connection name is optional. It represents the name of the job step that coordinates DB2 activities. If you do not specify this option, the connection name defaults are:

Type of Application
    Default Connection Name
Batch job
    Job name
Started task
    Started task name
TSO user
    TSO authorization ID

If a batch update job fails, you must use a separate job to restart the batch job. The connection name used in the restart job must be the same as the name used in the batch job that failed. Or, if the default connection name is used, the restart job must have the same job name as the batch update job that failed.

DB2 requires unique connection names. If two applications try to connect with the same connection name, then the second application program fails to connect to DB2.

The CONNECTION_NAME value can be from 1 to 8 characters long.

PLAN The DB2 plan name is optional. If you do not specify the plan name, then the application program module name is checked against the optional resource translation table. If there is a match in the resource translation table, the translated name is used as the DB2 plan name. If there is no match, then the application program module name is used as the plan name.

The PLAN value can be from 0 to 8 characters long.

PROG The application program name is required. It identifies the application program that is to be loaded and to receive control.

The PROG value can be from 1 to 8 characters long.

An example of the fields in the record is:

```
DSN,SYS1,DSNMIN10,,R,-,BATCH001,DB2PLAN,PROGA
```

DB2 DL/I batch output

In an online IMS environment, DB2 sends unsolicited status messages to the master terminal operator (MTO) and records on indoubt processing and diagnostic information to the IMS log. In a batch environment, DB2 sends this information to the output data set specified in the DDOTV02 DD statement. The output data set should have DCB options of RECFM=V or VB, LRECL=4092, and BLKSIZE of at least LRECL + 4. If the DD statement is missing, DB2 issues the message IEC130I and continues processing without any output.

You might want to save and print the data set, as the information is useful for diagnostic purposes. You can use the IMS module, DFSERA10, to print the variable-length data set records in both hexadecimal and character format.

Program preparation considerations

Consider the following as guidelines for program preparation when accessing DB2 and DL/I in a batch program.

Precompiling

When you add SQL statements to an application program, you must precompile the application program and bind the resulting DBRM into a plan or package, as described in "Chapter 20. Preparing an application program to run" on page 395.

Binding

The owner of the plan or package must have all the privileges required to execute the SQL statements embedded in it. Before a batch program can issue SQL statements, a DB2 plan must exist.

You can specify the plan name to DB2 in one of the following ways:
*   In the DDITV02 input data set.
*   In subsystem member specification.
*   By default; the plan name is then the application load module name specified in DDITV02.

DB2 passes the plan name to the IMS attach package. If you do not specify a plan name in DDITV02, and a resource translation table (RTT) does not exist or the name is not in the RTT, then DB2 uses the passed name as the plan name. If the name exists in the RTT, then the name translates to the plan specified for the RTT.

The recommended approach is to give the DB2 plan the same name as that of the application load module, which is the IMS attach default. The plan name must be the same as the program name.

Link-editing

DB2 has language interface routines for each unique supported environment. DB2 requires the IMS language interface routine for DL/I batch. It is also necessary to have DFSLI000 link-edited with the application program.

Loading and running

To run a program using DB2, you need a DB2 plan. The bind process creates the DB2 plan. DB2 first verifies whether the DL/I batch job step can connect to batch job DB2. Then DB2 verifies whether the application program can access DB2 and enforce user identification of batch jobs accessing DB2.

There are two ways to submit DL/I batch applications to DB2:

- The DL/I batch procedure can run module DSNMTV01 as the application program. DSNMTV01 loads the "real" application program. See "Submitting a DL/I batch application using DSNMTV01" for an example of JCL used to submit a DL/I batch application by this method.

- The DL/I batch procedure can run your application program without using module DSNMTV01. To accomplish this, do the following:

  - Specify SSM= in the DL/I batch procedure.

  - In the batch region of your application's JCL, specify the following:
    - MBR=*application-name*
    - SSM=*DB2 subsystem name*

  See "Submitting a DL/I batch application without using DSNMTV01" on page 484 for an example of JCL used to submit a DL/I batch application by this method.

Submitting a DL/I batch application using DSNMTV01

The following skeleton JCL example illustrates a COBOL application program, IVP8CP22, that runs using DB2 DL/I batch support.

- The first step uses the standard DLIBATCH IMS procedure.

- The second step shows how to use the DFSERA10 IMS program to print the contents of the DDOTV02 output data set.

```
//ISOCS04  JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
//         MSGCLASS=T,CLASS=A
//JOBLIB  DD  DISP=SHR,
//            DSN=prefix.SDSNLOAD
//* ****************************************************************
//*
//* THE FOLLOWING STEP SUBMITS COBOL JOB IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES.
//*
//* ****************************************************************
//UPDTE    EXEC DLIBATCH,DBRC=Y,LOGT=SYSDA,COND=EVEN,
// MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N
//G.STEPLIB  DD
//         DD
//         DD  DSN=prefix.SDSNLOAD,DISP=SHR
//         DD  DSN=prefix.RUNLIB.LOAD,DISP=SHR
//         DD  DSN=SYS1.COB2LIB,DISP=SHR
//         DD  DSN=IMS.PGMLIB,DISP=SHR
//G.STEPCAT  DD  DSN=IMSCAT,DISP=SHR
//G.DDOTV02  DD  DSN=&TEMP1,DISP=(NEW,PASS,DELETE),
//         SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
```

```
//         DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD  *
  SSDQ,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//***************************************************************
//***   ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET      ***
//***************************************************************
//STEP3    EXEC  PGM=DFSERA10,COND=EVEN
//STEPLIB   DD    DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    DSNAME=&TEMP1,DISP=(OLD,DELETE)
//SYSIN     DD    *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
//
```

Submitting a DL/I batch application without using DSNMTV01

The skeleton JCL in the following example illustrates a COBOL application program,
IVP8CP22, that runs using DB2 DL/I batch support.

```
//TEPCTEST JOB 'USER=ADMF001',MSGCLASS=A,MSGLEVEL=(1,1),
//         TIME=1440,CLASS=A,USER=SYSADM,PASSWORD=SYSADM
//****************************
//BATCH  EXEC DLIBATCH,PSB=IVP8CA,MBR=IVP8CP22,
//       BKO=Y,DBRC=N,IRLM=N,SSM=SSDQ
//****************************
//SYSPRINT DD SYSOUT=A
//REPORT   DD SYSOUT=*
//G.DDOTV02 DD DSN=&TEMP,DISP=(NEW,PASS,DELETE),
//    SPACE=(CYL,(10,1),RLSE),
//    UNIT=SYSDA,DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD *
SSDQ,SYS1,DSNMIN10,,Q,",DSNMTES1,,IVP8CP22
//G.SYSIN DD *
/*
//****************************************************
//*   ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET
//****************************************************
//PRTLOG EXEC PGM=DFSERA10,COND=EVEN
//STEPLIB   DD DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//SYSUT1    DD DSN=&TEMP,DISP=(OLD,DELETE)
//SYSIN     DD *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
```

Restart and recovery

To restart a batch program that updates data, you must first run the IMS batch
backout utility, followed by a restart job indicating the last successful checkpoint ID.
- Sample JCL for the utility is in "JCL example of a batch backout".
- Sample JCL for a restart job is in "JCL example of restarting a DL/I batch job" on
  page 485.
- For guidelines on finding the last successful checkpoint, see "Finding the DL/I
  batch checkpoint ID" on page 486.

JCL example of a batch backout

The skeleton JCL example that follows illustrates a batch backout for PSB=IVP8CA.

```
//ISOCS04  JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
//         MSGCLASS=T,CLASS=A
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
//*                                                              *
//*  BACKOUT TO LAST CHKPT.                                      *
//*                   IF RC=0028 LOG WITH NO-UPDATE              *
//*                                                              *
//* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
//BACKOUT  EXEC PGM=DFSRRC00,
//        PARM='DLI,DFSBB000,IVP8CA,,,,,,,,,,,,Y,N,,Y',
//         REGION=2600K,COND=EVEN            |
//*                                          ---> DBRC ON
//STEPLIB DD  DSN=IMS.RESLIB,DISP=SHR
//STEPCAT DD  DSN=IMSCAT,DISP=SHR
//IMS        DD   DSN=IMS.PSBLIB,DISP=SHR
//           DD   DSN=IMS.DBDLIB,DISP=SHR
//*
//* IMSLOGR  DD   data set is required
//* IEFRDER  DD   data set is required
//DFSVSAMP DD  *
OPTIONS,LTWA=YES
2048,7
1024,7
/*
//SYSIN DD DUMMY
/*
```

JCL example of restarting a DL/I batch job

Operational procedures can restart a DL/I batch job step for an application program using IMS XRST and symbolic CHKP calls.

You cannot restart A BMP application program in a DB2 DL/I batch environment. The symbolic checkpoint records are not accessed, causing an IMS user abend U0102.

To restart a batch job that terminated abnormally or prematurely, find the checkpoint ID for the job on the MVS system log or from the SYSOUT listing of the failing job. Before you restart the job step, place the checkpoint ID in the CKPTID=value option of the DLIBATCH procedure, then submit the job. If the default connection name is used (that is, you did not specify the connection name option in the DDITV02 input data set), the job name of the restart job must be the same as the failing job. Refer to the following skeleton example, in which the last checkpoint ID value was IVP80002:

```
//ISOCS04  JOB 3000,OJALA,MSGLEVEL=(1,1),NOTIFY=OJALA,
//        MSGCLASS=T,CLASS=A
//* ***************************************************************
//*
//* THE FOLLOWING STEP RESTARTS COBOL PROGRAM IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES, FROM CKPTID=IVP80002.
//*
//* ***************************************************************
//RSTRT    EXEC DLIBATCH,DBRC=Y,COND=EVEN,LOGT=SYSDA,
//  MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N,CKPTID=IVP80002
//G.STEPLIB  DD
//         DD
//         DD DSN=prefix.SDSNLOAD,DISP=SHR
//         DD DSN=prefix.RUNLIB.LOAD,DISP=SHR
//         DD DSN=SYS1.COB2LIB,DISP=SHR
//         DD DSN=IMS.PGMLIB,DISP=SHR
//*        other program libraries
//* G.IEFRDER  data set required
//G.STEPCAT  DD  DSN=IMSCAT,DISP=SHR
//* G.IMSLOGR  data set required
//G.DDOTV02  DD  DSN=&TEMP2,DISP=(NEW,PASS,DELETE),
//         SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
//         DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
```

```
//G.DDITV02  DD  *
  DB2X,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//****************************************************************
//***   ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET      ***
//****************************************************************
//STEP8      EXEC  PGM=DFSERA10,COND=EVEN
//STEPLIB    DD     DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT   DD     SYSOUT=A
//SYSUT1     DD     DSNAME=&TEMP2,DISP=(OLD,DELETE)
//SYSIN      DD     *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
//
```

Finding the DL/I batch checkpoint ID

When an application program issues an IMS CHKP call, IMS sends the checkpoint ID to the MVS console and the SYSOUT listing in message DFS0540I. IMS also records the checkpoint ID in the type X'41' IMS log record. Symbolic CHKP calls also create one or more type X'18' records on the IMS log. XRST uses the type X'18' log records to reposition DL/I databases and return information to the application program.

During the commit process the application program checkpoint ID is passed to DB2. If a failure occurs during the commit process, creating an indoubt work unit, DB2 remembers the checkpoint ID. You can use the following techniques to find the last checkpoint ID:

- Look at the SYSOUT listing for the job step to find message DFS0540I, which contains the checkpoint IDs issued. Use the last checkpoint ID listed.
- Look at the MVS console log to find message(s) DFS0540I containing the checkpoint ID issued for this batch program. Use the last checkpoint ID listed.
- Submit the IMS Batch Backout utility to back out the DL/I databases to the last (default) checkpoint ID. When the batch backout finishes, message DFS395I provides the last valid IMS checkpoint ID. Use this checkpoint ID on restart.
- When restarting DB2, the operator can issue the command -DISPLAY THREAD(*) TYPE(INDOUBT) to obtain a possible indoubt unit of work (connection name and checkpoint ID). If you restarted the application program from this checkpoint ID, it could work because the checkpoint is recorded on the IMS log; however, it could fail with an IMS user abend U102 because IMS did not finish logging the information before the failure. In that case, restart the application program from the previous checkpoint ID.

DB2 performs one of two actions automatically when restarted, if the failure occurs outside the indoubt period: it either backs out the work unit to the prior checkpoint, or it commits the data without any assistance. If the operator then issues the command

```
-DISPLAY THREAD(*) TYPE(INDOUBT)
```

no work unit information displays.

Part 6. Additional programming techniques

Chapter 23. Coding dynamic SQL in application programs

Before you decide to use dynamic SQL, you should consider whether using static SQL or dynamic SQL is the best technique for your application.

For most DB2 users, *static SQL*—embedded in a host language program and bound before the program runs—provides a straightforward, efficient path to DB2 data. You can use static SQL when you know before run time what SQL statements your application needs to execute.

Dynamic SQL prepares and executes the SQL statements within a program, while the program is running. There are four types of dynamic SQL:

- Embedded dynamic SQL

  Your application puts the SQL source in host variables and includes PREPARE and EXECUTE statements that tell DB2 to prepare and run the contents of those host variables at run time. You must precompile and bind programs that include embedded dynamic SQL.

- Interactive SQL

  A user enters SQL statements through SPUFI. DB2 prepares and executes those statements as dynamic SQL statements.

- Deferred embedded SQL

  Deferred embedded SQL statements are neither fully static nor fully dynamic. Like static statements, deferred embedded SQL statements are embedded within applications, but like dynamic statements, they are prepared at run time. DB2 processes deferred embedded SQL statements with bind-time rules. For example, DB2 uses the authorization ID and qualifier determined at bind time as the plan or package owner. Deferred embedded SQL statements are used for DB2 private protocol access to remote data.

- Dynamic SQL executed through ODBC functions

  Your application contains ODBC function calls that pass dynamic SQL statements as arguments. You do not need to precompile and bind programs that use ODBC function calls. See *DB2 ODBC Guide and Reference* for information on ODBC.

"Choosing between static and dynamic SQL" on page 496 suggests some reasons for choosing either static or dynamic SQL.

The rest of this chapter shows you how to code dynamic SQL in applications that contain three types of SQL statements:

- "Dynamic SQL for non-SELECT statements" on page 505. Those statements include DELETE, INSERT, and UPDATE.
- "Dynamic SQL for fixed-list SELECT statements" on page 509. A SELECT statement is *fixed-list* if you know in advance the number and type of data items in each row of the result.
- "Dynamic SQL for varying-list SELECT statements" on page 511. A SELECT statement is *varying-list* if you cannot know in advance how many data items to allow for or what their data types are.

Choosing between static and dynamic SQL

This section contains the following information to help you decide whether you should use dynamic SQL statements in your application:
- "Host variables make static SQL flexible"
- "Dynamic SQL is completely flexible"
- "What an application program using dynamic SQL does" on page 497
- "What dynamic SQL cannot do"
- "Performance of static and dynamic SQL" on page 497
- "Caching dynamic SQL statements" on page 498
- "Limiting dynamic SQL with the resource limit facility" on page 503
- "Choosing a host language for dynamic SQL applications" on page 504

Host variables make static SQL flexible

When you use static SQL, you cannot change the form of SQL statements unless you make changes to the program. However, you can increase the flexibility of those statements by using host variables.

In the example below, the UPDATE statement can update the salary of any employee. At bind time, you know that salaries must be updated, but you do not know until run time whose salaries should be updated, and by how much.

```
01  IOAREA.
    02  EMPID           PIC X(06).
    02  NEW-SALARY      PIC S9(7)V9(2) COMP-3.
:
 (Other declarations)
READ CARDIN RECORD INTO IOAREA
 AT END MOVE 'N' TO INPUT-SWITCH.
:
 (Other COBOL statements)
EXEC SQL
  UPDATE DSN8710.EMP
    SET SALARY = :NEW-SALARY
    WHERE EMPNO = :EMPID
END-EXEC.
```

The statement (UPDATE) does not change, nor does its basic structure, but the input can change the results of the UPDATE statement.

Dynamic SQL is completely flexible

What if a program must use different types and structures of SQL statements? If there are so many types and structures that it cannot contain a model of each one, your program might need dynamic SQL.

One example of such a program is the Query Management Facility (QMF), which provides an alternative interface to DB2 that accepts almost any SQL statement. SPUFI is another example; it accepts SQL statements from an input data set, and then processes and executes them dynamically.

What dynamic SQL cannot do

You can use only some of the SQL statements dynamically. For information on which DB2 SQL statements you can dynamically prepare, see the table in "Appendix G. Characteristics of SQL statements in DB2 for OS/390 and z/OS" on page 919.

What an application program using dynamic SQL does

A program that provides for dynamic SQL accepts as input, or generates, an SQL statement in the form of a character string. You can simplify the programming if you can plan the program not to use SELECT statements, or to use only those that return a known number of values of known types. In the most general case, in which you do not know in advance about the SQL statements that will execute, the program typically takes these steps:

1. Translates the input data, including any parameter markers, into an SQL statement
2. Prepares the SQL statement to execute and acquires a description of the result table
3. Obtains, for SELECT statements, enough main storage to contain retrieved data
4. Executes the statement or fetches the rows of data
5. Processes the information returned
6. Handles SQL return codes.

Performance of static and dynamic SQL

To access DB2 data, an SQL statement requires an access path. Two big factors in the performance of an SQL statement are the amount of time that DB2 uses to determine the access path at run time and whether the access path is efficient. DB2 determines the access path for a statement at either of these times:
- When you bind the plan or package that contains the SQL statement
- When the SQL statement executes

The time at which DB2 determines the access path depends on these factors:
- Whether the statement is executed statically or dynamically
- Whether the statement contains input host variables

Static SQL statements with no input host variables
For static SQL statements that do not contain input host variables, DB2 determines the access path when you bind the plan or package. This combination yields the best performance because the access path is already determined when the program executes.

Static SQL statements with input host variables
For these statements, the time at which DB2 determines the access path depends on whether you specify the bind option NOREOPT(VARS) or REOPT(VARS). NOREOPT(VARS) is the default.

If you specify NOREOPT(VARS), DB2 determines the access path at bind time, just as it does when there are no input variables.

If you specify REOPT(VARS), DB2 determines the access path at bind time and again at run time, using the values in these types of input variables:
- Host variables
- Parameter markers
- Special registers

This means that DB2 must spend extra time determining the access path for statements at run time, but if DB2 determines a significantly better access path using the variable values, you might see an overall performance improvement. In general, using REOPT(VARS) can make static SQL statements with input variables

perform like dynamic SQL statements with constants. For more information about using REOPT(VARS) to change access paths, see "Using host variables efficiently" on page 644.

Dynamic SQL statements

For dynamic SQL statements, DB2 determines the access path at run time, when the statement is prepared. This can make the performance worse than that of static SQL statements. However, if you execute the same SQL statement often, you can use the dynamic statement cache to decrease the number of times that those dynamic statements must be prepared. See "Performance of static and dynamic SQL" on page 497 for more information.

Dynamic SQL statements with input host variables: In general, it is recommended that you use the option REOPT(VARS) when you bind applications that contain dynamic SQL statements with input host variables. However, you should code your PREPARE statements to minimize overhead. With REOPT(VARS), DB2 prepares an SQL statement at the same time as it processes OPEN or EXECUTE for the statement. That is, DB2 processes the statement as if you specified DEFER(PREPARE). However, if you execute the DESCRIBE statement before the PREPARE statement in your program, or if you use the PREPARE statement with the INTO parameter, DB2 prepares the statement twice. The first time, DB2 determines the access path without using input variable values, and the second time DB2 uses the input variable values. The extra prepare can decrease your performance. For a statement that uses a cursor, you can avoid the double prepare by placing the DESCRIBE statement after the OPEN statement in your program.

If you use predictive governing, and a dynamic SQL statement bound with REOPT(VARS) exceeds a predictive governing warning threshold, your application does not receive a warning SQLCODE. However, if the statement exceeds a predictive governing error threshold, the application receives an error SQLCODE from the OPEN or EXECUTE statement.

Caching dynamic SQL statements

As DB2's ability to optimize SQL has improved, the cost of preparing a dynamic SQL statement has grown. Applications that use dynamic SQL might be forced to pay this cost more than once. When an application performs a commit operation, it must issue another PREPARE statement if that SQL statement is to be executed again. For a SELECT statement, the ability to declare a cursor WITH HOLD provides some relief but requires that the cursor be open at the commit point. WITH HOLD also causes some locks to be held for any objects that the prepared statement is dependent on. Also, WITH HOLD offers no relief for SQL statements that are not SELECT statements.

DB2 can save prepared dynamic statements in a cache. The cache is a DB2-wide cache in the EDM pool that all application processes can use to store and retrieve prepared dynamic statements. After an SQL statement has been prepared and is automatically stored in the cache, subsequent prepare requests for that same SQL statement can avoid the costly preparation process by using the statement in the cache. Cached statements can be shared among different threads, plans, or packages.

For example:

```
PREPARE STMT1 FROM ...    Statement is prepared and the prepared
EXECUTE STMT1             statement is put in the cache.
COMMIT
.
.
.
PREPARE STMT1 FROM ...    Identical statement. DB2 uses the prepared
EXECUTE STMT1             statement from the cache.
COMMIT
.
.
.
```

Eligible Statements: The following SQL statements are eligible for caching:
- SELECT
- UPDATE
- INSERT
- DELETE

Distributed and local SQL statements are eligible. Prepared, dynamic statements using DB2 private protocol access are eligible.

Restrictions: Even though static statements that use DB2 private protocol access are dynamic at the remote site, those statements are not eligible for caching.

Statements in plans or packages bound with REOPT(VARS) are not eligible for caching. See "How bind option REOPT(VARS) affects dynamic SQL" on page 523 for more information about REOPT(VARS).

Prepared statements cannot be shared among data sharing members. Because each member has its own EDM pool, a cached statement on one member is not available to an application that runs on another member.

Using the dynamic statement cache

To enable caching of prepared statements, specify YES on the CACHE DYNAMIC SQL field of installation panel DSNTIP4. See Part 2 of *DB2 Installation Guide* for more information.

Conditions for statement sharing

Suppose that S1 and S2 are source statements, and P1 is the prepared version of S1. P1 is in the prepared statement cache.

The following conditions must be met before DB2 can use statement P1 instead of preparing statement S2:

- S1 and S2 must be identical. The statements must pass a character by character comparison and must be the same length. If the PREPARE statement for either statement contains an ATTRIBUTES clause, DB2 concatenates the values in the ATTRIBUTES clause to the statement string before comparing the strings. That is, if A1 is the set of attributes for S1 and A2 is the set of attributes for S2, DB2 compares S1||A1 to S2||A2.

  If the statement strings are not identical, DB2 cannot use the statement in the cache.

  For example, if S1 and S2 are both

  `'UPDATE EMP SET SALARY=SALARY+50'`

  then DB2 can use P1 instead of preparing S2. However, if S1 is

  `'UPDATE EMP SET SALARY=SALARY+50'`

  and S2 is

```
'UPDATE EMP SET SALARY=SALARY+50 '
```

then DB2 cannot use P1.

In that case, DB2 prepares S2 and puts the prepared version of S2 in the cache.
- The authorization ID that was used to prepare S1 must be used to prepare S2:
  – When a plan or package has run behavior, the authorization ID is the current SQLID value.

    For secondary authorization IDs:
    - The application process that searches the cache must have the same secondary authorization ID list as the process that inserted the entry into the cache or must have a superset of that list.
    - If the process that originally prepared the statement and inserted it into the cache used one of the privileges held by the primary authorization ID to accomplish the prepare, that ID must either be part of the secondary authorization ID list of the process searching the cache, or it must be the primary authorization ID of that process.
  – When a plan or package has bind behavior, the authorization ID is the plan owner's ID. For a DDF server thread, the authorization ID is the package owner's ID.
  – When a package has define behavior, then the authorization ID is the user-defined function or stored procedure owner.
  – When a package has invoke behavior, then the authorization ID is the authorization ID under which the statement that invoked the user-defined function or stored procedure executed.

  For an explanation of bind, run, define, and invoke behavior, see "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415.
- When the plan or package that contains S2 is bound, the values of these bind options must be the same as when the plan or package that contains S1 was bound:
    CURRENTDATA
    DYNAMICRULES
    ISOLATION
    SQLRULES
    QUALIFIER
- When S2 is prepared, the values of special registers CURRENT DEGREE, CURRENT RULES, and CURRENT PRECISION must be the same as when S1 was prepared.

Keeping prepared statements after commit points

The bind option KEEPDYNAMIC(YES) lets you hold dynamic statements past a commit point for an application process. An application can issue a PREPARE for a statement once and omit subsequent PREPAREs for that statement. Figure 146 on page 501 illustrates an application that is written to use KEEPDYNAMIC(YES).

```
PREPARE STMT1 FROM ...    Statement is prepared.
EXECUTE STMT1
COMMIT
:
:
EXECUTE STMT1             Application does not issue PREPARE.
COMMIT
:
:
EXECUTE STMT1             Again, no PREPARE needed.
COMMIT
```

Figure 146. Writing dynamic SQL to use the bind option KEEPDYNAMIC(YES)

To understand how the KEEPDYNAMIC bind option works, it is important to differentiate between the executable form of a dynamic SQL statement, the *prepared statement*, and the character string form of the statement, the *statement string*.

Relationship between KEEPDYNAMIC(YES) and statement caching: When the dynamic statement cache is not active, and you run an application bound with KEEPDYNAMIC(YES), DB2 saves only the statement string for a prepared statement after a commit operation. On a subsequent OPEN, EXECUTE, or DESCRIBE, DB2 must prepare the statement again before performing the requested operation. Figure 147 illustrates this concept.

```
PREPARE STMT1 FROM ...    Statement is prepared and put in memory.
EXECUTE STMT1
COMMIT
:
:
EXECUTE STMT1             Application does not issue PREPARE.
COMMIT                    DB2 prepares the statement again.
:
:
EXECUTE STMT1             Again, no PREPARE needed.
COMMIT
```

Figure 147. Using KEEPDYNAMIC(YES) when the dynamic statement cache is not active

When the dynamic statement cache is active, and you run an application bound with KEEPDYNAMIC(YES), DB2 retains a copy of both the prepared statement and the statement string. The prepared statement is cached locally for the application process. It is likely that the statement is globally cached in the EDM pool, to benefit other application processes. If the application issues an OPEN, EXECUTE, or DESCRIBE after a commit operation, the application process uses its local copy of the prepared statement to avoid a prepare and a search of the cache. Figure 148 on page 502 illustrates this process.

```
PREPARE STMT1 FROM ...     Statement is prepared and put in memory.
EXECUTE STMT1
COMMIT
  .
  .
EXECUTE STMT1              Application does not issue PREPARE.
COMMIT                     DB2 uses the prepared statement in memory.
  .
  .
EXECUTE STMT1              Again, no PREPARE needed.
COMMIT                     DB2 uses the prepared statement in memory.
  .
  .
PREPARE STMT1 FROM ...     Statement is prepared and put in memory.
```

Figure 148. Using KEEPDYNAMIC(YES) when the dynamic statement cache is active

The local instance of the prepared SQL statement is kept in *ssnm*DBM1 storage until one of the following occurs:

- The application process ends.
- A rollback operation occurs.
- The application issues an explicit PREPARE statement with the same statement name.

  If the application does issue a PREPARE for the same SQL statement name that has a kept dynamic statement associated with it, the kept statement is discarded and DB2 prepares the new statement.

- The statement is removed from memory because the statement has not been used recently, and the number of kept dynamic SQL statements reaches a limit set at installation time.

Handling implicit prepare errors: If a statement is needed during the lifetime of an application process, and the statement has been removed from the local cache, DB2 might be able to retrieve it from the global cache. If the statement is not in the global cache, DB2 must implicitly prepare the statement again. The application does not need to issue a PREPARE statement. However, if the application issues an OPEN, EXECUTE, or DESCRIBE for the statement, the application must be able to handle the possibility that DB2 is doing the prepare *implicitly*. Any error that occurs during this prepare is returned on the OPEN, EXECUTE, or DESCRIBE.

How KEEPDYNAMIC affects applications that use distributed data: If a requester does not issue a PREPARE after a COMMIT, the package at the DB2 for OS/390 and z/OS server must be bound with KEEPDYNAMIC(YES). If both requester and server are DB2 for OS/390 and z/OS subsystems, the DB2 requester assumes that the KEEPDYNAMIC value for the package at the server is the same as the value for the plan at the requester.

The KEEPDYNAMIC option has performance implications for DRDA clients that specify WITH HOLD on their cursors:

- If KEEPDYNAMIC(NO) is specified, a separate network message is required when the DRDA client issues the SQL CLOSE for the cursor.
- If KEEPDYNAMIC(YES) is specified, the DB2 for OS/390 and z/OS server automatically closes the cursor when SQLCODE +100 is detected, which means that the client does not have to send a separate message to close the held cursor. This reduces network traffic for DRDA applications that use held cursors. It also reduces the duration of locks that are associated with the held cursor.

Using RELEASE(DEALLOCATE) with KEEPDYNAMIC(YES): See "The RELEASE option and dynamic statement caching" on page 338 for information about interactions between bind options RELEASE(DEALLOCATE) and KEEPDYNAMIC(YES).

Considerations for data sharing: If one member of a data sharing group has enabled the cache but another has not, and an application is bound with KEEPDYNAMIC(YES), DB2 must implicitly prepare the statement again if the statement is assigned to a member without the cache. This can mean a slight reduction in performance.

Limiting dynamic SQL with the resource limit facility

The resource limit facility (or governor) limits the amount of CPU time an SQL statement can take, which prevents SQL statements from making excessive requests. The predictive governing function of the resource limit facility provides an estimate of the processing cost of SQL statements before they run. To predict the cost of an SQL statement, you execute EXPLAIN to put information about the statement cost in DSN_STATEMNT_TABLE. See "Estimating a statement's cost" on page 711 for information on creating, populating, and interpreting the contents of DSN_STATEMNT_TABLE.

The governor controls only the dynamic SQL manipulative statements SELECT, UPDATE, DELETE, and INSERT. Each dynamic SQL statement used in a program is subject to the same limits. The limit can be a *reactive* governing limit or a *predictive* governing limit. If the statement exceeds a reactive governing limit, the statement receives an error SQL code. If the statement exceeds a predictive governing limit, it receives a warning or error SQL code. "Writing an application to handle predictive governing" explains more about predictive governing SQL codes.

Your system administrator can establish the limits for individual plans or packages, for individual users, or for all users who do not have personal limits.

Follow the procedures defined by your location for adding, dropping, or modifying entries in the resource limit specification table. For more information on the resource limit specification tables, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Writing an application to handle reactive governing

When a dynamic SQL statement exceeds a reactive governing threshold, the application program receives SQLCODE -905. The application must then determine what to do next.

If the failed statement involves an SQL cursor, the cursor's position remains unchanged. The application can then close that cursor. All other operations with the cursor do not run and the same SQL error code occurs.

If the failed SQL statement does not involve a cursor, then all changes that the statement made are undone before the error code returns to the application. The application can either issue another SQL statement or commit all work done so far.

Writing an application to handle predictive governing

If your installation uses predictive governing, you need to modify your applications to check for the +495 and -495 SQLCODEs that predictive governing can generate after a PREPARE statement executes. The +495 SQLCODE in combination with

deferred prepare requires that DB2 do some special processing to ensure that existing applications are not affected by this new warning SQLCODE.

For information about setting up the resource limit facility for predictive governing, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Handling the +495 SQLCODE

If your requester uses deferred prepare, the presence of parameter markers determines when the application receives the +495 SQLCODE. When parameter markers are present, DB2 cannot do PREPARE, OPEN, and FETCH processing in one message. If SQLCODE +495 is returned, no OPEN or FETCH processing occurs until your application requests it.

- If there are parameter markers, the +495 is returned on the OPEN (not the PREPARE).
- If there are no parameter markers, the +495 is returned on the PREPARE.

Normally with deferred prepare, the PREPARE, OPEN, and first FETCH of the data are returned to the requester. For a predictive governor warning of +495, you would ideally like to have the option to choose beforehand whether you want the OPEN and FETCH of the data to occur. For downlevel requesters, you do not have this option.

Using predictive governing and downlevel DRDA requesters

If SQLCODE +495 is returned to the requester, OPEN processing continues but the first block of data is not returned with the OPEN. Thus, if your application does not continue with the query, you have already incurred the performance cost of OPEN processing.

Using predictive governing and enabled requesters

If your application does not defer the prepare, SQLCODE +495 is returned to the requester and OPEN processing does not occur.

If your application does defer prepare processing, the application receives the +495 at its usual time (OPEN or PREPARE). If you have parameter markers with deferred prepare, you receive the +495 at OPEN time as you normally do. However, an additional message is exchanged.

Recommendation: Do not use deferred prepare for applications that use parameter markers and that are predictively governed at the server side.

Choosing a host language for dynamic SQL applications

Programs that use dynamic SQL are usually written in assembler, C, PL/I, REXX, and versions of COBOL other than OS/VS COBOL. You can write non-SELECT and fixed-list SELECT statements in any of the DB2 supported languages. A program containing a varying-list SELECT statement is more difficult to write in FORTRAN, because the program cannot run without the help of a subroutine to manage address variables (pointers) and storage allocation.

All SQL in REXX programs is dynamic SQL. For information on how to write SQL REXX applications, see "Coding SQL statements in a REXX application" on page 189

Most of the examples in this section are in PL/I. "Using dynamic SQL in COBOL" on page 524 shows techniques for using COBOL. Longer examples in the form of complete programs are available in the sample applications:

DSNTEP2
> Processes both SELECT and non-SELECT statements dynamically. (PL/I).

DSNTIAD
> Processes only non-SELECT statements dynamically. (Assembler).

DSNTIAUL
> Processes SELECT statements dynamically. (Assembler).

Library *prefix*.SDSNSAMP contains the sample programs. You can view the programs online, or you can print them using ISPF, IEBPTPCH, or your own printing program.

Dynamic SQL for non-SELECT statements

The easiest way to use dynamic SQL is not to use SELECT statements dynamically. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including OS/VS COBOL and FORTRAN. For a sample program written in C that contains dynamic SQL with non-SELECT statements, refer to Figure 244 on page 859.

Your program must take the following steps:

1. Include an SQLCA. The requirements for an SQL communications area (SQLCA) are the same as for static SQL statements. For REXX, DB2 includes the SQLCA automatically.

2. Load the input SQL statement into a data area. The procedure for building or reading the input SQL statement is not discussed here; the statement depends on your environment and sources of information. You can read in complete SQL statements, or you can get information to build the statement from data sets, a user at a terminal, previously set program variables, or tables in the database.

    If you attempt to execute an SQL statement dynamically that DB2 does not allow, you get an SQL error.

3. Execute the statement. You can use either of these methods:
    - "Dynamic execution using EXECUTE IMMEDIATE"
    - "Dynamic execution using PREPARE and EXECUTE" on page 506.

4. Handle any errors that might result. The requirements are the same as those for static SQL statements. The return code from the most recently executed SQL statement appears in the host variables SQLCODE and SQLSTATE or corresponding fields of the SQLCA. See "Checking the execution of SQL statements" on page 74 for information on the SQLCA and the fields it contains.

Dynamic execution using EXECUTE IMMEDIATE

Suppose you design a program to read SQL DELETE statements, similar to these, from a terminal:

```
DELETE FROM DSN8710.EMP WHERE EMPNO = '000190'
DELETE FROM DSN8710.EMP WHERE EMPNO = '000220'
```

After reading a statement, the program is to execute it immediately.

Recall that you must prepare (precompile and bind) static SQL statements before you can use them. You cannot prepare dynamic SQL statements in advance. The SQL statement EXECUTE IMMEDIATE causes an SQL statement to prepare and execute, dynamically, at run time.

The EXECUTE IMMEDIATE statement

To execute the statements:

```
< Read a DELETE statement into the host variable DSTRING.>
EXEC SQL
  EXECUTE IMMEDIATE :DSTRING;
```

DSTRING is a character-string host variable. EXECUTE IMMEDIATE causes the DELETE statement to be prepared and executed immediately.

The host variable

DSTRING is the name of a host variable, and is not a DB2 reserved word. In assembler, COBOL and C, you must declare it as a varying-length string variable. In FORTRAN, it must be a fixed-length string variable. In PL/I, it can be a fixed- or varying-length character string variable, or any PL/I expression that evaluates to a character string. For more information on varying-length string variables, see "Chapter 9. Embedding SQL statements in host languages" on page 107.

Dynamic execution using PREPARE and EXECUTE

Suppose that you want to execute DELETE statements repeatedly using a list of employee numbers. Consider how you would do it if you could write the DELETE statement as a static SQL statement:

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
  EXEC SQL
    DELETE FROM DSN8710.EMP WHERE EMPNO = :EMP ;
  < Read a value for EMP from the list. >
END;
```

The loop repeats until it reads an EMP value of 0.

If you know in advance that you will use only the DELETE statement and only the table DSN8710.EMP, then you can use the more efficient static SQL. Suppose further that there are several different tables with rows identified by employee numbers, and that users enter a table name as well as a list of employee numbers to delete. Although variables can represent the employee numbers, they cannot represent the table name, so you must construct and execute the entire statement dynamically. Your program must now do these things differently:
- Use parameter markers instead of host variables
- Use the PREPARE statement
- Use EXECUTE instead of EXECUTE IMMEDIATE.

Using parameter markers

Dynamic SQL statements cannot use host variables. Therefore, you cannot dynamically execute an SQL statement that contains host variables. Instead, substitute a *parameter marker*, indicated by a question mark (?), for each host variable in the statement.

You can indicate to DB2 that a parameter marker represents a host variable of a certain data type by specifying the parameter marker as the argument of a CAST function. When the statement executes, DB2 converts the host variable to the data

type in the CAST function. A parameter marker that you include in a CAST function is called a *typed* parameter marker. A parameter marker without a CAST function is called an *untyped* parameter marker.

Because DB2 can evaluate an SQL statement with typed parameter markers more efficiently than a statement with untyped parameter markers, we recommend that you use typed parameter markers whenever possible. Under certain circumstances you must use typed parameter markers. See Chapter 5 of *DB2 SQL Reference* for rules for using untyped or typed parameter markers.

Example: To prepare this statement:

```
DELETE FROM DSN8710.EMP WHERE EMPNO = :EMP;
```

prepare a string like this:

```
DELETE FROM DSN8710.EMP WHERE EMPNO = CAST(? AS CHAR(6))
```

You associate host variable :EMP with the parameter marker when you execute the prepared statement. Suppose S1 is the prepared statement. Then the EXECUTE statement looks like this:

```
EXECUTE S1 USING :EMP;
```

The PREPARE statement

You can think of PREPARE and EXECUTE as an EXECUTE IMMEDIATE done in two steps. The first step, PREPARE, turns a character string into an SQL statement, and then assigns it a name of your choosing.

For example, let the variable :DSTRING have the value "DELETE FROM DSN8710.EMP WHERE EMPNO = ?". To prepare an SQL statement from that string and assign it the name S1, write:

```
EXEC SQL PREPARE S1 FROM :DSTRING;
```

The prepared statement still contains a parameter marker, for which you must supply a value when the statement executes. After the statement is prepared, the table name is fixed, but the parameter marker allows you to execute the same statement many times with different values of the employee number.

The EXECUTE statement

EXECUTE executes a prepared SQL statement, naming a list of one or more host variables, or a host structure, that supplies values for all of the parameter markers.

After you prepare a statement, you can execute it many times within the same unit of work. In most cases, COMMIT or ROLLBACK destroys statements prepared in a unit of work. Then, you must prepare them again before you can execute them again. However, if you declare a cursor for a dynamic statement and use the option WITH HOLD, a commit operation does not destroy the prepared statement if the cursor is still open. You can execute the statement in the next unit of work without preparing it again.

To execute the prepared statement S1 just once, using a parameter value contained in the host variable :EMP, write:

```
EXEC SQL EXECUTE S1 USING :EMP;
```

The complete example

The example began with a DO loop that executed a static SQL statement repeatedly:

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
  EXEC SQL
    DELETE FROM DSN8710.EMP WHERE EMPNO = :EMP ;
  < Read a value for EMP from the list. >
END;
```

You can now write an equivalent example for a dynamic SQL statement:

```
< Read a statement containing parameter markers into DSTRING.>
EXEC SQL PREPARE S1 FROM :DSTRING;
< Read a value for EMP from the list. >
DO UNTIL (EMPNO = 0);
  EXEC SQL EXECUTE S1 USING :EMP;
  < Read a value for EMP from the list. >
  END;
```

The PREPARE statement prepares the SQL statement and calls it S1. The EXECUTE statement executes S1 repeatedly, using different values for EMP.

More than one parameter marker

The prepared statement (S1 in the example) can contain more than one parameter marker. If it does, the USING clause of EXECUTE specifies a list of variables or a host structure. The variables must contain values that match the number and data types of parameters in S1 in the proper order. You must know the number and types of parameters in advance and declare the variables in your program, or you can use an SQLDA (SQL descriptor area).

Using DESCRIBE INPUT to put parameter marker information in the SQLDA

You can use the DESCRIBE INPUT statement to let DB2 put the data type information for parameter markers in an SQLDA.

Before you execute DESCRIBE INPUT, you must allocate an SQLDA with enough instances of SQLVAR to represent all parameter markers in the SQL statements you want to describe.

After you execute DESCRIBE INPUT, you code the application in the same way as any other application in which you execute a prepared statement using an SQLDA. First, you obtain the addresses of the input host variables and their indicator variables and insert those addresses into the SQLDATA and SQLIND fields. Then you execute the prepared SQL statement.

For example, suppose you want to execute this statement dynamically:

```
DELETE FROM DSN8710.EMP WHERE EMPNO = ?
```

The code to set up an SQLDA, obtain parameter information using DESCRIBE INPUT, and execute the statement looks like this:

```
SQLDAPTR=ADDR(INSQLDA);        /* Get pointer to SQLDA       */
SQLDAID='SQLDA';               /* Fill in SQLDA eye-catcher  */
SQLDABC=LENGTH(INSQLDA);       /* Fill in SQLDA length       */
SQLN=1;                        /* Fill in number of SQLVARs  */
SQLD=0;                        /* Initialize # of SQLVARs used */
DO IX=1 TO SQLN;               /* Initialize the SQLVAR      */
  SQLTYPE(IX)=0;
  SQLLEN(IX)=0;
  SQLNAME(IX)='';
END;
SQLSTMT='DELETE FROM DSN8710.EMP WHERE EMPNO = ?';
EXEC SQL PREPARE SQLOBJ FROM SQLSTMT;
EXEC SQL DESCRIBE INPUT SQLOBJ INTO :INSQLDA;
```

```
SQLDATA(1)=ADDR(HVEMP);          /* Get input data address    */
SQLIND(1)=ADDR(HVEMPIND);        /* Get indicator address     */
EXEC SQL EXECUTE SQLOBJ USING DESCRIPTOR :INSQLDA;
```

Dynamic SQL for fixed-list SELECT statements

A *fixed-list* SELECT statement returns rows containing a known number of values of a known type. When you use one, you know in advance exactly what kinds of host variables you need to declare in order to store the results. (The contrasting situation, in which you do *not* know in advance what host-variable structure you might need, is in the section "Dynamic SQL for varying-list SELECT statements" on page 511.)

The term "fixed-list" does not imply that you must know in advance how many rows of data will return; however, you must know the number of columns and the data types of those columns. A fixed-list SELECT statement returns a result table that can contain any number of rows; your program looks at those rows one at a time, using the FETCH statement. Each successive fetch returns the same number of values as the last, and the values have the same data types each time. Therefore, you can specify host variables as you do for static SQL.

An advantage of the fixed-list SELECT is that you can write it in any of the programming languages that DB2 supports. Varying-list dynamic SELECT statements require assembler, C, PL/I, and versions of COBOL other than OS/VS COBOL.

For a sample program written in C illustrating dynamic SQL with fixed-list SELECT statements, see Figure 244 on page 859.

What your application program must do

To execute a fixed-list SELECT statement dynamically, your program must:
1. Include an SQLCA
2. Load the input SQL statement into a data area

   The preceding two steps are exactly the same as described under "Dynamic SQL for non-SELECT statements" on page 505.
3. Declare a cursor for the statement name as described in "Declare a cursor for the statement name" on page 510.
4. Prepare the statement, as described in "Prepare the statement" on page 510.
5. Open the cursor, as described in "Open the cursor" on page 510.
6. Fetch rows from the result table, as described in "Fetch rows from the result table" on page 510.
7. Close the cursor, as described in "Close the cursor" on page 511.
8. Handle any resulting errors. This step is the same as for static SQL, except for the number and types of errors that can result.

Suppose that your program retrieves last names and phone numbers by dynamically executing SELECT statements of this form:
```
SELECT LASTNAME, PHONENO FROM DSN8710.EMP
  WHERE ... ;
```

The program reads the statements from a terminal, and the user determines the WHERE clause.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Eventually you prepare a statement from DSTRING, but first you must declare a cursor for the statement and give it a name.

Declare a cursor for the statement name

Dynamic SELECT statements cannot use INTO; hence, you must use a cursor to put the results into host variables. In declaring the cursor, use the statement name (call it STMT), and give the cursor itself a name (for example, C1):

```
EXEC SQL DECLARE C1 CURSOR FOR STMT;
```

Prepare the statement

Prepare a statement (STMT) from DSTRING. This is one possible PREPARE statement:

```
EXEC SQL PREPARE STMT FROM :DSTRING ATTRIBUTES :ATTRVAR;
```

ATTRVAR contains attributes that you want to add to the SELECT statement, such as FETCH FIRST 10 ROWS ONLY or OPTIMIZE for 1 ROW. In general, if the SELECT statement has attributes that conflict with the attributes in the PREPARE statement, the attributes on the SELECT statement take precedence over the attributes on the PREPARE statement. However, in this example, the SELECT statement in DSTRING has no attributes specified, so DB2 uses the attributes in ATTRVAR for the SELECT statement.

As with non-SELECT statements, the fixed-list SELECT could contain parameter markers. However, this example does not need them.

To execute STMT, your program must open the cursor, fetch rows from the result table, and close the cursor. The following sections describe how to do those steps.

Open the cursor

The OPEN statement evaluates the SELECT statement named STMT. For example:

Without parameter markers: EXEC SQL OPEN C1;

If STMT contains parameter markers, then you must use the USING clause of OPEN to provide values for all of the parameter markers in STMT. If there are four parameter markers in STMT, you need:

```
EXEC SQL OPEN C1 USING :PARM1, :PARM2, :PARM3, :PARM4;
```

Fetch rows from the result table

Your program could repeatedly execute a statement such as this:

```
EXEC SQL FETCH C1 INTO :NAME, :PHONE;
```

The key feature of this statement is the use of a list of host variables to receive the values returned by FETCH. The list has a known number of items (two—:NAME and :PHONE) of known data types (both are character strings, of lengths 15 and 4, respectively).

It is possible to use this list in the FETCH statement only because you planned the program to use only fixed-list SELECTs. Every row that cursor C1 points to must contain exactly two character values of appropriate length. If the program is to handle anything else, it must use the techniques described under Dynamic SQL for varying-list SELECT statements.

Close the cursor

This step is the same as for static SQL. For example, a WHENEVER NOT FOUND statement in your program can name a routine that contains this statement:

```
EXEC SQL CLOSE C1;
```

Dynamic SQL for varying-list SELECT statements

A *varying-list* SELECT statement returns rows containing an unknown number of values of unknown type. When you use one, you do *not* know in advance exactly what kinds of host variables you need to declare in order to store the results. (For the much simpler situation, in which you *do* know, see "Dynamic SQL for fixed-list SELECT statements" on page 509.) Because the varying-list SELECT statement requires pointer variables for the SQL descriptor area, you cannot issue it from a FORTRAN or an OS/VS COBOL program. A FORTRAN or OS/VS COBOL program can call a subroutine written in a language that supports pointer variables (such as PL/I or assembler), if you need to use a varying-list SELECT statement.

What your application program must do

To execute a varying-list SELECT statement dynamically, your program must follow these steps:

1. Include an SQLCA

   DB2 performs this step for a REXX procedure.

2. Load the input SQL statement into a data area

   Those first two steps are exactly the same as described under "Dynamic SQL for non-SELECT statements" on page 505; the next step is new:

3. Prepare and execute the statement. This step is more complex than for fixed-list SELECTs. For details, see "Preparing a varying-list SELECT statement" and "Executing a varying-list SELECT statement dynamically" on page 521. It involves the following steps:

   a. Include an SQLDA (SQL descriptor area).

      DB2 performs this step for a REXX procedure.

   b. Declare a cursor and prepare the variable statement.

   c. Obtain information about the data type of each column of the result table.

   d. Determine the main storage needed to hold a row of retrieved data.

      You do not perform this step for a REXX procedure.

   e. Put storage addresses in the SQLDA to tell where to put each item of retrieved data.

   f. Open the cursor.

   g. Fetch a row.

   h. Eventually close the cursor and free main storage.

   There are further complications for statements with parameter markers.

4. Handle any errors that might result.

Preparing a varying-list SELECT statement

Suppose your program dynamically executes SQL statements, but this time without any limits on their form. Your program reads the statements from a terminal, and you know nothing about them in advance. They might not even be SELECT statements.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Your program goes on to prepare a statement from the variable and then give the statement a name; call it STMT.

Now there is a new wrinkle. The program must find out whether the statement is a SELECT. If it is, the program must also find out how many values are in each row, and what their data types are. The information comes from an *SQL descriptor area* (SQLDA).

An SQL descriptor area (SQLDA)

The SQLDA is a structure used to communicate with your program, and storage for it is usually allocated dynamically at run time.

To include the SQLDA in a PL/I or C program, use:

```
EXEC SQL INCLUDE SQLDA;
```

For assembler, use this in the storage definition area of a CSECT:

```
EXEC SQL INCLUDE SQLDA
```

For COBOL, except for OS/VS COBOL, use:

```
EXEC SQL INCLUDE SQLDA END-EXEC.
```

You cannot include an SQLDA in an OS/VS COBOL, FORTRAN, or REXX program.

For a complete layout of the SQLDA and the descriptions given by INCLUDE statements, see Appendix C of *DB2 SQL Reference*.

Obtaining information about the SQL statement

An SQLDA can contain a variable number of occurrences of SQLVAR, each of which is a set of five fields that describe one column in the result table of a SELECT statement.

The number of occurrences of SQLVAR depends on the following factors:
- The number of columns in the result table you want to describe.
- Whether you want the PREPARE or DESCRIBE to put both column names and labels in your SQLDA. This is the option USING BOTH in the PREPARE or DESCRIBE statement.
- Whether any columns in the result table are LOB types or distinct types.

Table 56 shows the minimum number of SQLVAR instances you need for a result table that contains *n* columns.

Table 56. Minimum number of SQLVARs for a result table with n columns

Type of DESCRIBE and contents of result table	Not USING BOTH	USING BOTH
No distinct types or LOBs	*n*	2*n
Distinct types but no LOBs	2*n	3*n
LOBs but no distinct types	2*n	2*n
LOBs and distinct types	2*n	3*n

We call an SQLDA with *n* occurrences of SQLVAR a *single SQLDA*, an SQLDA with 2*n occurrences of SQLVAR a *double SQLDA*, an SQLDA with 3*n occurrences of SQLVAR a *triple SQLDA*.

A program that admits SQL statements of every kind for dynamic execution has two choices:

- Provide the largest SQLDA that it could ever need. The maximum number of columns in a result table is 750, so an SQLDA for 750 columns occupies 33 016 bytes for a single SQLDA, 66 016 bytes for a double SQLDA, or 99 016 bytes for a triple SQLDA. Most SELECTs do not retrieve 750 columns, so the program does not usually use most of that space.

- Provide a smaller SQLDA, with fewer occurrences of SQLVAR. From this the program can find out whether the statement was a SELECT and, if it was, how many columns are in its result table. If there are more columns in the result than the SQLDA can hold, DB2 returns no descriptions. When this happens, the program must acquire storage for a second SQLDA that is long enough to hold the column descriptions, and ask DB2 for the descriptions again. Although this technique is more complicated to program than the first, it is more general.

  How many columns should you allow? You must choose a number that is large enough for most of your SELECT statements, but not too wasteful of space; 40 is a good compromise. To illustrate what you must do for statements that return more columns than allowed, the example in this discussion uses an SQLDA that is allocated for at least 100 columns.

Declaring a cursor for the statement

As before, you need a cursor for the dynamic SELECT. For example, write:

```
EXEC SQL
  DECLARE C1 CURSOR FOR STMT;
```

Preparing the statement using the minimum SQLDA

Suppose your program declares an SQLDA structure with the name MINSQLDA, having 100 occurrences of SQLVAR and SQLN set to 100. To prepare a statement from the character string in DSTRING and also enter its description into MINSQLDA, write this:

```
EXEC SQL PREPARE STMT FROM :DSTRING;
EXEC SQL DESCRIBE STMT INTO :MINSQLDA;
```

Equivalently, you can use the INTO clause in the PREPARE statement:

```
EXEC SQL
  PREPARE STMT INTO :MINSQLDA FROM :DSTRING;
```

Do not use the USING clause in either of these examples. At the moment, only the minimum SQLDA is in use. Figure 149 shows the contents of the minimum SQLDA in use.

Header ⟶	SQLDAID	SQLDABC	100	SQLD

Figure 149. The minimum SQLDA structure

SQLn determines what SQLVAR gets

The SQLN field, which you must set before using DESCRIBE (or PREPARE INTO), tells how many occurrences of SQLVAR the SQLDA is allocated for. If DESCRIBE needs more than that, the results of the DESCRIBE depend on the contents of the result table. Let n indicate the number of columns in the result table. Then:

- If the result table contains at least one distinct type column but no LOB columns, you do not specify USING BOTH, and $n<=SQLN<2*n$, then DB2 returns base SQLVAR information in the first n SQLVAR occurrences, but no distinct type information. Base SQLVAR information includes:

- – Data type code
- – Length attribute (except for LOBs)
- – Column name or label
- – Host variable address
- – Indicator variable address
- Otherwise, if SQLN is less than the minimum number of SQLVARs specified in Table 56 on page 512, then DB2 returns no information in the SQLVARs.

Whether or not your SQLDA is big enough, whenever you execute DESCRIBE, DB2 returns the following values, which you can use to build an SQLDA of the correct size:

- SQLD

  0 if the SQL statement is not a SELECT. Otherwise, the number of columns in the result table. The number of SQLVAR occurrences you need for the SELECT depends on the value in the 7th byte of SQLDAID.
- The 7th byte of SQLDAID

  2 if each column in the result table requires 2 SQLVAR entries. 3 if each column in the result table requires 3 SQLVAR entries.

If the statement is not a SELECT

To find out if the statement is a SELECT, your program can query the SQLD field in MINSQLDA. If the field contains 0, the statement is *not* a SELECT, the statement is already prepared, and your program can execute it. If there are no parameter markers in the statement, you can use:

```
EXEC SQL EXECUTE STMT;
```

(If the statement does contain parameter markers, you must use an SQL descriptor area; for instructions, see "Executing arbitrary statements with parameter markers" on page 522.)

Acquiring storage for a second SQLDA if needed

Now you can allocate storage for a second, full-size SQLDA; call it FULSQLDA. Figure 150 on page 515 shows its structure.

FULSQLDA has a fixed-length header of 16 bytes in length, followed by a varying-length section that consists of structures with the SQLVAR format. If the result table contains LOB columns or distinct type columns, a varying-length section that consists of structures with the SQLVAR2 format follows the structures with SQLVAR format. All SQLVAR structures and SQLVAR2 structures are 44 bytes long. See Appendix C of *DB2 SQL Reference* for details on the two SQLVAR formats. The number of SQLVAR and SQLVAR2 elements you need is in the SQLD field of MINSQLDA, and the total length you need for FULSQLDA (16 + SQLD * 44) is in the SQLDABC field of MINSQLDA. Allocate that amount of storage.
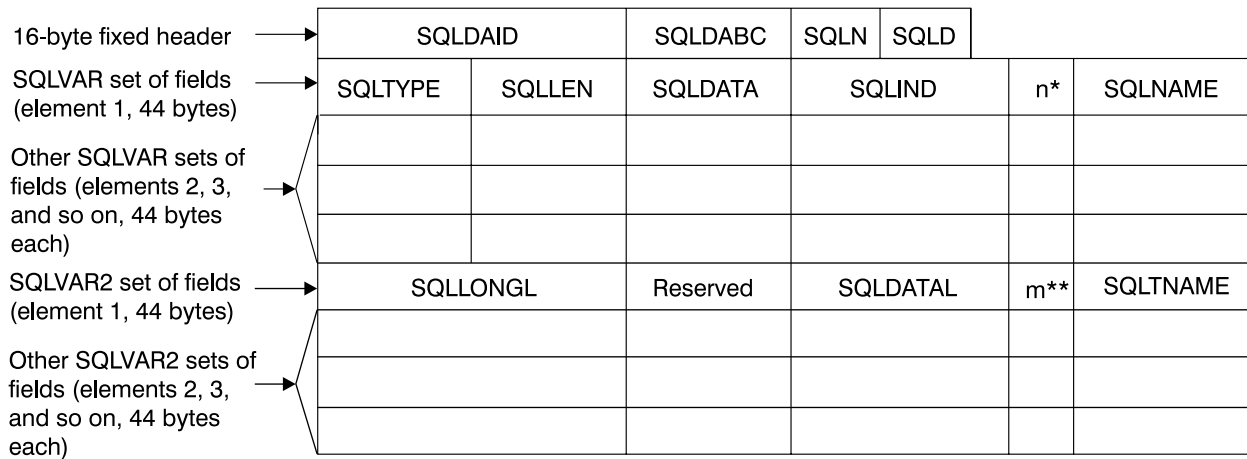
Figure 150. The SQLDA structure

Describing the SELECT statement again

Having allocated sufficient space for FULSQLDA, your program must take these steps:

1. Put the total number of SQLVAR and SQLVAR2 occurrences in FULSQLDA into the SQLN field of FULSQLDA. This number appears in the SQLD field of MINSQLDA.

2. Describe the statement again into the new SQLDA:

```
EXEC SQL DESCRIBE STMT INTO :FULSQLDA;
```

After the DESCRIBE statement executes, each occurrence of SQLVAR in the full-size SQLDA (FULSQLDA in our example) contains a description of one column of the result table in five fields. If an SQLVAR occurrence describes a LOB column or distinct type column, the corresponding SQLVAR2 occurrence contains additional information specific to the LOB or distinct type. Figure 151 shows an SQLDA that describes two columns that are not LOB columns or distinct type columns. See "Describing tables with LOB and distinct type columns" on page 519 for an example of describing a result table with LOB columns or distinct type columns.
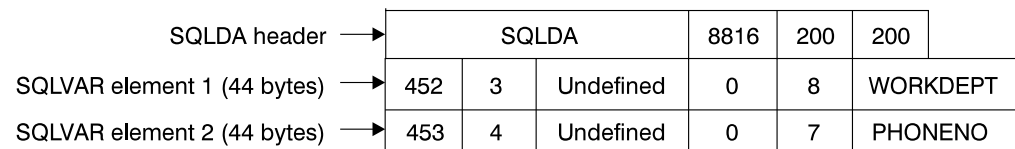
Figure 151. Contents of FULSQLDA after executing DESCRIBE

Acquiring storage to hold a row

Before fetching rows of the result table, your program must:

1. Analyze each SQLVAR description to determine how much space you need for the column value.
2. Derive the address of some storage area of the required size.
3. Put this address in the SQLDATA field.

If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field.

Figure 152, Figure 153, and Figure 154 on page 517 show the SQL descriptor area after you take certain actions. Table 57 on page 517 describes the values in the descriptor area. In Figure 152, the DESCRIBE statement inserted all the values *except* the first occurrence of the number 200. The program inserted the number 200 before it executed DESCRIBE to tell how many occurrences of SQLVAR to allow. If the result table of the SELECT has more columns than this, the SQLVAR fields describe nothing.

The next set of five values, the first SQLVAR, pertains to the first column of the result table (the WORKDEPT column). SQLVAR element 1 contains fixed-length character strings and does not allow null values (SQLTYPE=452); the length attribute is 3. For information on SQLTYPE values, see Appendix C of *DB2 SQL Reference*.

SQLDA header →	SQLDA		8816	200	200	
SQLVAR element 1 (44 bytes) →	452	3	Undefined	0	8	WORKDEPT
SQLVAR element 2 (44 bytes) →	453	4	Undefined	0	7	PHONENO

Figure 152. SQL descriptor area after executing DESCRIBE

SQLDA header →	SQLDA		8816	200	200	
SQLVAR element 1 (44 bytes) →	452	3	Addr FLDA	Addr FLDAI	8	WORKDEPT
SQLVAR element 2 (44 bytes) →	453	4	Addr FLDB	Addr FLDBI	7	PHONENO

FLDA
CHAR(3)

FLDB
CHAR(4)

Indicator variables (halfword)
FLDAI   FLDBI

Figure 153. SQL descriptor area after analyzing descriptions and acquiring storage
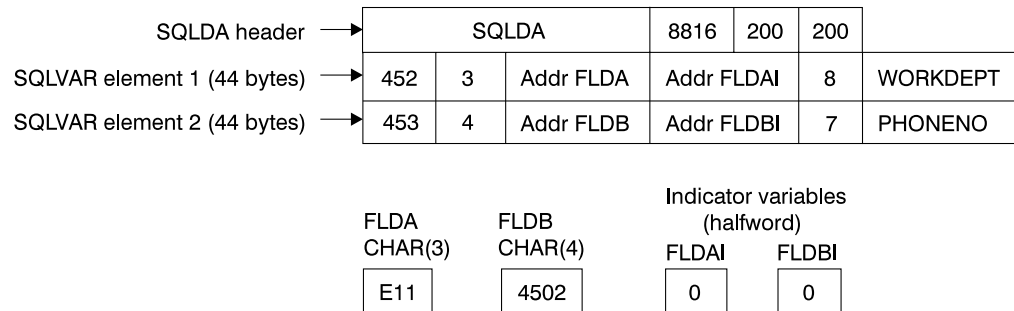
Figure 154. SQL descriptor area after executing FETCH

Table 57. Values inserted in the SQLDA

Value	Field	Description
SQLDA	SQLDAID	An "eye-catcher"
8816	SQLDABC	The size of the SQLDA in bytes (16 + 44 * 200)
200	SQLN	The number of occurrences of SQLVAR, set by the program
200	SQLD	The number of occurrences of SQLVAR actually used by the DESCRIBE statement
452	SQLTYPE	The value of SQLTYPE in the first occurrence of SQLVAR. It indicates that the first column contains fixed-length character strings, and does not allow nulls.
3	SQLLEN	The length attribute of the column
Undefined or CCSID value	SQLDATA	Bytes 3 and 4 contain the CCSID of a string column. Undefined for other types of columns.
Undefined	SQLIND	
8	SQLNAME	The number of characters in the column name
WORKDEPT	SQLNAME+2	The column name of the first column

Putting storage addresses in the SQLDA

After analyzing the description of each column, your program must replace the content of each SQLDATA field with the address of a storage area large enough to hold values from that column. Similarly, for every column that allows nulls, the program must replace the content of the SQLIND field. The content must be the address of a halfword that you can use as an indicator variable for the column. The program can acquire storage for this purpose, of course, but the storage areas used do not have to be contiguous.

Figure 153 on page 516 shows the content of the descriptor area before the program obtains any rows of the result table. Addresses of fields and indicator variables are already in the SQLVAR.

Changing the CCSID for retrieved data

All DB2 string data, if it is not defined with FOR BIT DATA, has an encoding scheme and CCSID associated with it. When you select string data from a table, the selected data generally has the same encoding scheme and CCSID as the table, with one exception: If you perform a query against a DB2 for OS/390 and z/OS table that is defined with CCSID ASCII or CCSID UNICODE, the retrieved

data is encoded in the application encoding scheme for the DB2 subsystem. The application encoding scheme is the value that was specified in the APPLICATION ENCODING field of installation panel DSNTIPF.

In dynamic SQL statements, if you want to retrieve the data in an encoding scheme and CCSID other than the default values, you can use one of the following techniques:

- Set the CURRENT APPLICATION ENCODING SCHEME special register before you execute the SELECT statements. For example, to set the CCSID and encoding scheme for retrieved data to the default CCSID for Unicode, execute this SQL statement:

  ```
  EXEC SQL SET CURRENT APPLICATION ENCODING SCHEME ='UNICODE';
  ```

- For fixed-list SELECT statements, use the DECLARE VARIABLE statement to associate CCSIDs with the host variables into which you retrieve the data. See "Changing the coded character set ID of host variables" on page 72 for information on this technique.

- For varying-list SELECT statements, set the CCSID for the retrieved data in the SQLDA. The following text describes that technique.

To change the encoding scheme of retrieved data, set up the SQLDA as you would for any other varying-list SELECT statement. Then make these additional changes to the SQLDA:

1. Put the character + in the sixth byte of field SQLDAID.

2. For each SQLVAR entry:
   - Set the length field of SQLNAME to 8.
   - Set the first two bytes of the data field of SQLNAME to X'0000'.
   - Set the third and fourth bytes of the data field of SQLNAME to the CCSID, in hexadecimal, in which you want the results to display. You can specify any CCSID that meets either of the following conditions:
     - There is a row in catalog table SYSSTRINGS that has a matching value for OUTCCSID.
     - Language Environment supports conversion to that CCSID. See OS/390 C/C++ Programming Guide for information on the conversions that Language Environment supports.

     If you are modifying the CCSID to retrieve the contents of an ASCII or Unicode table on a DB2 for OS/390 and z/OS system, and you previously executed a DESCRIBE statement on the SELECT statement that you are using to retrieve the data, the SQLDATA fields in the SQLDA that you used for the DESCRIBE contain the ASCII or Unicode CCSID for that table. To set the data portion of the SQLNAME fields for the SELECT, move the contents of each SQLDATA field in the SQLDA from the DESCRIBE to each SQLNAME field in the SQLDA for the SELECT. If you are using the same SQLDA for the DESCRIBE and the SELECT, be sure to move the contents of the SQLDATA field to SQLNAME before you modify the SQLDATA field for the SELECT.

     For REXX, you set the CCSID in the *stem.n.*SQLCCSID field instead of setting the SQLDAID and SQLNAME fields.

For example, suppose the table that contains WORKDEPT and PHONENO is defined with CCSID ASCII. To retrieve data for columns WORKDEPT and

PHONENO in ASCII CCSID 437 (X'01B5'), change the SQLDA as shown in Figure 155.

SQLDA header →	SQLDA+		8816	200	200
SQLVAR element 1 (44 bytes) →	452	3	Addr FLDA	Addr FLDAI	8	X'000001B500000000'
SQLVAR element 2 (44 bytes) →	453	4	Addr FLDB	Addr FLDBI	8	X'000001B500000000'

FLDA CHAR(3) FLDB CHAR(4) Indicator variables (halfword) FLDAI FLDBI
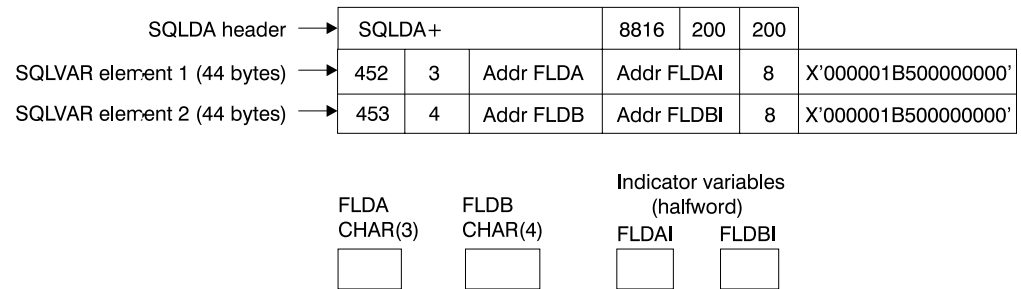
Figure 155. SQL descriptor area for retrieving data in ASCII CCSID 437

Using column labels

By default, DESCRIBE describes each column in the SQLNAME field by the column name. You can tell it to use column labels instead by writing:

```
EXEC SQL
  DESCRIBE STMT INTO :FULSQLDA USING LABELS;
```

In this case, SQLNAME contains nothing for a column with no label. If you prefer to use labels wherever they exist, but column names where there are no labels, write USING ANY. (Some columns, such as those derived from functions or expressions, have neither name nor label; SQLNAME contains nothing for those columns. However, if the column is the result of a UNION, SQLNAME contains the names of the columns of the first operand of the UNION.)

You can also write USING BOTH to obtain the name and the label when both exist. However, to obtain both, you need a second set of occurrences of SQLVAR in FULSQLDA. The first set contains descriptions of all the columns using names; the second set contains descriptions using labels. This means that you must allocate a longer SQLDA for the second DESCRIBE statement ((16 + SQLD * 88 bytes) instead of (16 + SQLD * 44)). You must also put double the number of columns (SLQD * 2) in the SQLN field of the second SQLDA. Otherwise, if there is not enough space available, DESCRIBE does not enter descriptions of any of the columns.

Describing tables with LOB and distinct type columns

In general, the steps you perform when you prepare an SQLDA to select rows from a table with LOB or distinct type columns are similar to the steps you perform if the table has no columns of this type. The only difference is that you need to analyze some additional fields in the SQLDA for LOB or distinct type columns.

To illustrate this, suppose you want to execute this SELECT statement:

```
SELECT USER, A_DOC FROM DOCUMENTS;
```

USER cannot contain nulls and is of distinct type ID, defined like this:

```
CREATE DISTINCT TYPE SCHEMA1.ID AS CHAR(20);
```

and A_DOC can contain nulls and is of type CLOB(1M).

The result table for this statement has two columns, but you need four SQLVAR occurrences in your SQLDA because the result table contains a LOB type and a distinct type. Suppose you prepare and describe this statement into FULSQLDA,

which is large enough to hold four SQLVAR occurrences. FULSQLDA looks like Figure 156.

SQLDA header →	SQLDA 2		192	4	4	
SQLVAR element 1 (44 bytes) →	452	20	Undefined	0	4	USER
SQLVAR element 2 (44 bytes) →	409	0	Undefined	0	5	A_DOC
SQLVAR2 element 1 (44 bytes) →					7	SCH1.ID
SQLVAR2 element 2 (44 bytes) →	1 048 576				11	SYSIBM.CLOB

Figure 156. SQL descriptor area after describing a CLOB and distinct type

The next steps are the same as for result tables without LOBs or distinct types:

1. Analyze each SQLVAR description to determine the maximum amount of space you need for the column value.

   For a LOB type, retrieve the length from the SQLLONGL field instead of the SQLLEN field.

2. Derive the address of some storage area of the required size.

   For a LOB data type, you also need a 4-byte storage area for the length of the LOB data. You can allocate this 4-byte area at the beginning of the LOB data or in a different location.

3. Put this address in the SQLDATA field.

   For a LOB data type, if you allocated a separate area to hold the length of the LOB data, put the address of the length field in SQLDATAL. If the length field is at beginning of the LOB data area, put 0 in SQLDATAL.

4. If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field.

Figure 157 and Figure 158 on page 521 show the contents of FULSQLDA after you fill in pointers to storage locations and execute FETCH.

SQLDA header →	SQLDA 2		192	4	4	
SQLVAR element 1 (44 bytes) →	452	20	Addr FLDA	0	4	USER
SQLVAR element 2 (44 bytes) →	409	0	Addr FLDB	Addr FLDBI	5	A_DOC
SQLVAR2 element 1 (44 bytes) →					7	SCH1.ID
SQLVAR2 element 2 (44 bytes) →	1 048 576			0	11	SYSIBM.CLOB

FLDA
CHAR(20)

FLDB
CLOB(1M)

Indicator Variable
(halfword)
FLDBI

4-byte
length
field
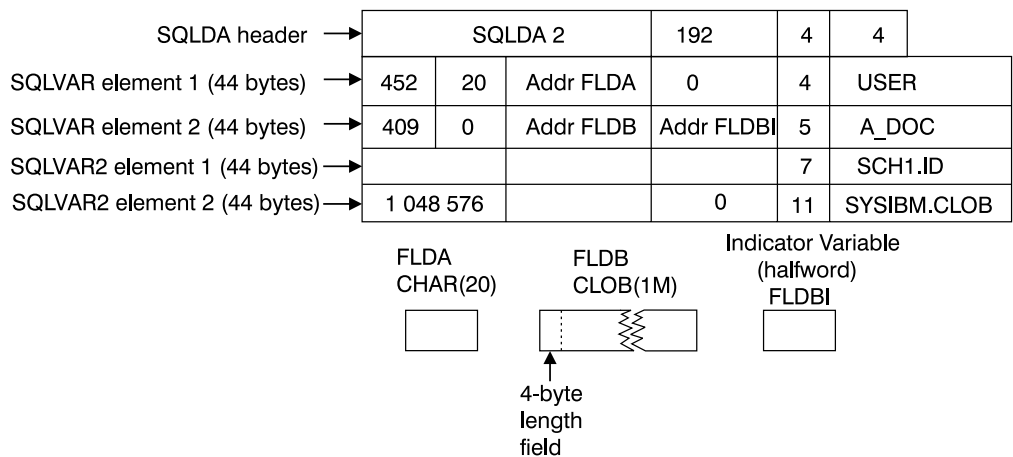
Figure 157. SQL descriptor area after analyzing CLOB and distinct type descriptions and acquiring storage

SQLDA header →	SQLDA 2		192	4	4	
SQLVAR element 1 (44 bytes) →	452	20	Addr FLDA	0	4	USER
SQLVAR element 2 (44 bytes) →	409	0	Addr FLDB	Addr FLDBI	5	A_DOC
SQLVAR2 element 1 (44 bytes) →					7	SCH1.ID
SQLVAR2 element 2 (44 bytes) →	1 048 576			0	11	SYSIBM.CLOB

FLDA
CHAR(20)

USR1

FLDB
CLOB(1M)

To: ......

Indicator Variable
(halfword)
FLDBI

0

4-byte length
field contains
actual length
of data
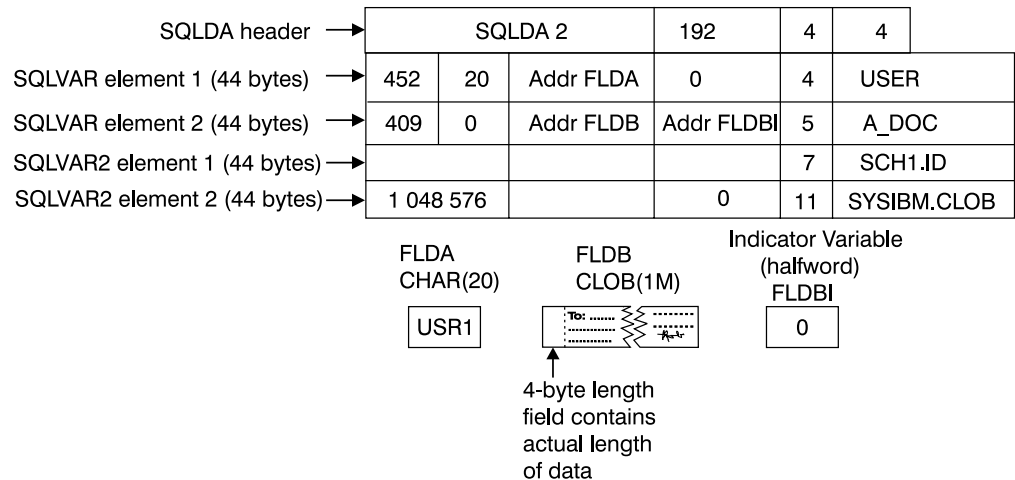
Figure 158. SQL descriptor area after executing FETCH on a table with CLOB and distinct type columns

Executing a varying-list SELECT statement dynamically

You can easily retrieve rows of the result table using a varying-list SELECT statement. The statements differ only a little from those for the fixed-list example.

Open the cursor

If the SELECT statement contains no parameter marker, this step is simple enough. For example:

```
EXEC SQL OPEN C1;
```

For cases when there are parameter markers, see "Executing arbitrary statements with parameter markers" on page 522 below.

Fetch rows from the result table

This statement differs from the corresponding one for the case of a fixed-list select. Write:

```
EXEC SQL
  FETCH C1 USING DESCRIPTOR :FULSQLDA;
```

The key feature of this statement is the clause USING DESCRIPTOR :FULSQLDA. That clause names an SQL descriptor area in which the occurrences of SQLVAR point to other areas. Those other areas receive the values that FETCH returns. It is possible to use that clause only because you previously set up FULSQLDA to look like Figure 152 on page 516.

Figure 154 on page 517 shows the result of the FETCH. The data areas identified in the SQLVAR fields receive the values from a single row of the result table.

Successive executions of the same FETCH statement put values from successive rows of the result table into these same areas.

Close the cursor

This step is the same as for the fixed-list case. When there are no more rows to process, execute the following statement:

```
EXEC SQL CLOSE C1;
```

When COMMIT ends the unit of work containing OPEN, the statement in STMT reverts to the unprepared state. Unless you defined the cursor using the WITH HOLD option, you must prepare the statement again before you can reopen the cursor.

Executing arbitrary statements with parameter markers

Consider, as an example, a program that executes dynamic SQL statements of several kinds, including varying-list SELECT statements, any of which might contain a variable number of parameter markers. This program might present your users with lists of choices: choices of operation (update, select, delete); choices of table names; choices of columns to select or update. The program also allows the users to enter lists of employee numbers to apply to the chosen operation. From this, the program constructs SQL statements of several forms, one of which looks like this:

```
SELECT .... FROM DSN8710.EMP
  WHERE EMPNO IN (?,?,?,...?);
```

The program then executes these statements dynamically.

When the number and types of parameters are known

In the above example, you do not know in advance the number of parameter markers, and perhaps the kinds of parameter they represent. You can use techniques described previously if you know the number and types of parameters, as in the following examples:

- If the SQL statement *is not* SELECT, name a list of host variables in the EXECUTE statement:

  WRONG: EXEC SQL EXECUTE STMT;

  RIGHT: EXEC SQL EXECUTE STMT USING :VAR1, :VAR2, :VAR3;

- If the SQL statement *is* SELECT, name a list of host variables in the OPEN statement:

  WRONG: EXEC SQL OPEN C1;

  RIGHT: EXEC SQL OPEN C1 USING :VAR1, :VAR2, :VAR3;

In *both* cases, the number and types of host variables named must agree with the number of parameter markers in STMT and the types of parameter they represent. The first variable (VAR1 in the examples) must have the type expected for the first parameter marker in the statement, the second variable must have the type expected for the second marker, and so on. There must be at least as many variables as parameter markers.

When the number and types of parameters are not known

When you do not know the number and types of parameters, you can adapt the SQL descriptor area. There is no limit to the number of SQLDAs your program can include, and you can use them for different purposes. Suppose an SQLDA, arbitrarily named DPARM, describes a set of parameters.

The structure of DPARM is the same as that of any other SQLDA. The number of occurrences of SQLVAR can vary, as in previous examples. In this case, there must be one for every parameter marker. Each occurrence of SQLVAR describes one host variable that replaces one parameter marker at run time. This happens either when a non-SELECT statement executes or when a cursor is opened for a SELECT statement.

You must fill in certain fields in DPARM *before* using EXECUTE or OPEN; you can ignore the other fields.

Field   Use When Describing Host Variables for Parameter Markers

SQLDAID
>   The seventh byte indicates whether more than one SQLVAR entry is used
>   for each parameter marker. If this byte is not blank, at least one parameter
>   marker represents a distinct type or LOB value, so the SQLDA has more
>   than one set of SQLVAR entries.
>
>   You do not set this field for a REXX SQLDA.

SQLDABC
>   The length of the SQLDA, equal to SQLN * 44 + 16. You do not set this
>   field for a REXX SQLDA.

SQLN   The number of occurrences of SQLVAR allocated for DPARM. You do not
>   set this field for a REXX SQLDA.

SQLD   The number of occurrences of SQLVAR actually used. This must not be
>   less than the number of parameter markers. In each occurrence of
>   SQLVAR, put the following information using the same way that you use the
>   DESCRIBE statement:

SQLTYPE
>   The code for the type of variable, and whether it allows nulls

SQLLEN
>   The length of the host variable

SQLDATA
>   The address of the host variable.
>
>   For REXX, this field contains the value of the host variable.

SQLIND
>   The address of an indicator variable, if needed.
>
>   For REXX, this field contains a negative number if the value in SQLDATA is
>   null.

SQLNAME
>   Ignored

Using the SQLDA with EXECUTE or OPEN

To indicate that the SQLDA called DPARM describes the host variables substituted
for the parameter markers at run time, use a USING DESCRIPTOR clause with
EXECUTE or OPEN.

- For a non-SELECT statement, write:

```
EXEC SQL EXECUTE STMT USING DESCRIPTOR :DPARM;
```

- For a SELECT statement, write:

```
EXEC SQL OPEN C1 USING DESCRIPTOR :DPARM;
```

How bind option REOPT(VARS) affects dynamic SQL

When you specify the bind option REOPT(VARS), DB2 reoptimizes the access path
at run time for SQL statements that contain host variables, parameter markers, or
special registers. The option REOPT(VARS) has the following effects on dynamic
SQL statements:

- When you specify the option REOPT(VARS), DB2 automatically uses
  DEFER(PREPARE), which means that DB2 waits to prepare a statement until it
  encounters an OPEN or EXECUTE statement.

- When you execute a DESCRIBE statement and then an EXECUTE statement on
  a non-SELECT statement, DB2 prepares the statement twice: Once for the
  DESCRIBE statement and once for the EXECUTE statement. DB2 uses the
  values in the input variables only during the second PREPARE. These multiple
  PREPAREs can cause performance to degrade if your program contains many
  dynamic non-SELECT statements. To improve performance, consider putting the

code that contains those statements in a separate package and then binding that package with the option NOREOPT(VARS).

- If you execute a DESCRIBE statement before you open a cursor for that statement, DB2 prepares the statement twice. If, however, you execute a DESCRIBE statement after you open the cursor, DB2 prepares the statement only once. To improve the performance of a program bound with the option REOPT(VARS), execute the DESCRIBE statement *after* you open the cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.

- If you use predictive governing for applications bound with REOPT(VARS), DB2 does not return a warning SQL code when dynamic SQL statements exceed the predictive governing warning threshold. DB2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. DB2 returns the error SQL code for an EXECUTE or OPEN statement.

Using dynamic SQL in COBOL

You can use all forms of dynamic SQL in all versions of COBOL except OS/VS COBOL. OS/VS COBOL programs using an SQLDA must use an assembler subroutine to manage address variables (pointers) and to allocate storage. For a detailed description and a working example of the method, see "Sample COBOL dynamic SQL program" on page 845.

Chapter 24. Using stored procedures for client/server processing

This chapter covers the following topics:
- "Introduction to stored procedures"
- "An example of a simple stored procedure" on page 526
- "Setting up the stored procedures environment" on page 530
- "Writing and preparing an external stored procedure" on page 537
- "Writing and preparing an SQL procedure" on page 552
- "Writing and preparing an application to use stored procedures" on page 569
- "Running a stored procedure" on page 610
- "Testing a stored procedure" on page 615

This chapter contains information that applies to all stored procedures and specific information about stored procedures in languages other than Java. For information on writing, preparing, and running Java stored procedures, see *DB2 Application Programming Guide and Reference for Java*.

Introduction to stored procedures

A *stored procedure* is a compiled program, stored at a DB2 local or remote server, that can execute SQL statements. A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language. A client application program uses the SQL statement CALL to invoke the stored procedure.

Consider using stored procedures for a client/server application that does at least one of the following things:
- Executes many remote SQL statements.

  Remote SQL statements can create many network send and receive operations, which results in increased processor costs.

  Stored procedures can encapsulate many of your application's SQL statements into a single message to the DB2 server, reducing network traffic to a single send and receive operation for a series of SQL statements.
- Accesses host variables for which you want to guarantee security and integrity.

  Stored procedures remove SQL applications from the workstation, which prevents workstation users from manipulating the contents of sensitive SQL statements and host variables.

Figure 159 on page 526 and Figure 160 on page 526 illustrate the difference between using stored procedures and not using stored procedures.
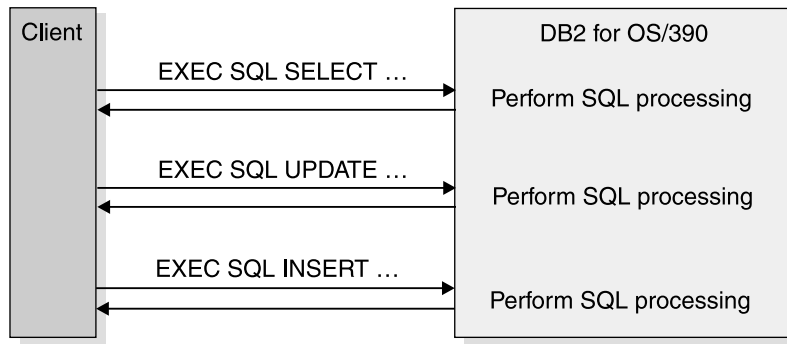
Figure 159. Processing without stored procedures. An application embeds SQL statements and communicates with the server separately for each one.
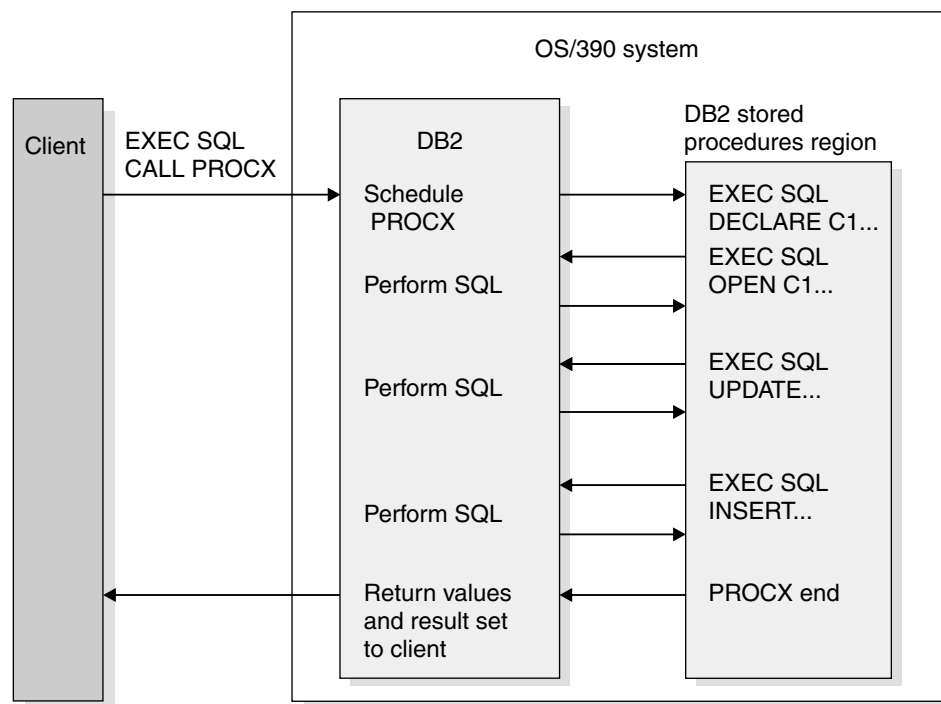
Figure 160. Processing with stored procedures. The same series of SQL statements uses a single send or receive operation.

An example of a simple stored procedure

Suppose that an application runs on a workstation client and calls a stored procedure *A* on the DB2 server at location LOCA. Stored Procedure *A* performs the following operations:

1. Receives a set of parameters containing the data for one row of the employee to project activity table (DSN8710.EMPPROJACT). These parameters are input parameters in the SQL statement CALL:
   - EMP: employee number
   - PRJ: project number
   - ACT: activity ID
   - EMT: percent of employee's time required
   - EMS: date the activity starts
   - EME: date the activity is due to end

2. Declares a cursor, C1, with the option WITH RETURN, that is used to return a result set containing all rows in EMPPROJACT to the caller.

3. Queries table EMPPROJACT to determine whether a row exists where columns PROJNO, ACTNO, EMSTDATE, and EMPNO match the values of parameters PRJ, ACT, EMS, and EMP. (The table has a unique index on those columns. There is at most one row with those values.)

4. If the row exists, executes an SQL statement UPDATE to assign the values of parameters EMT and EME to columns EMPTIME and EMENDATE.

5. If the row does not exist, executes an SQL statement INSERT to insert a new row with all the values in the parameter list.

6. Opens cursor C1. This causes the result set to be returned to the caller when the stored procedure ends.

7. Returns two parameters, containing these values:
   - A code to identify the type of SQL statement last executed: UPDATE or INSERT.
   - The SQLCODE from that statement.

Notes   User Workstation   DB2 System   DB2 Stored Procedures Address Space

Stored Procedure A

1 EXEC SQL CONNECT TO LOCA;

2 Create Thread

3 EXEC SQL CALL A(:EMP, :PRJ,:ACT,:EMT, :EMS,:EME, :TYPE,:CODE);

4 Get information from SYSIBM. SYSROUTINES

5 Prepare parameter list and pass control to stored procedure

6 EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR SELECT * FROM EMPPROJACT;

USE SQL UPDATE to update EMPPROJACT with input parameter values

If SQLCODE=+100, use SQL INSERT to add a row with the values in the parameter list

EXEC SQL OPEN C1;

7 Return output parameters :TYPE and :CODE and a result set that contains all rows in EMPPROJACT

8 Control returns to application

9 EXEC SQL COMMIT; (or ROLLBACK)

Result of COMMIT or ROLLBACK

10 Receive result set

Figure 161. Stored procedure overview

Notes to Figure 161:

1. The workstation application uses the SQL CONNECT statement to create a conversation with DB2.
2. DB2 creates a DB2 thread to process SQL requests.

3. The SQL statement CALL tells the DB2 server that the application is going to run a stored procedure. The calling application provides the necessary parameters.

4. The plan for the client application contains information from catalog table SYSROUTINES about stored procedure A. DB2 caches all rows in the table associated with A, so future references to A do not require I/O to the table.

5. DB2 passes information about the request to the stored procedures address space, and the stored procedure begins execution.

6. The stored procedure executes SQL statements.

   DB2 verifies that the owner of the package or plan containing the SQL statement CALL has EXECUTE authority for the package associated with the DB2 stored procedure.

   One of the SQL statements opens a cursor that has been declared WITH RETURN. This causes a result set to be returned to the workstation application.

7. The stored procedure assigns values to the output parameters and exits. Control returns to the DB2 stored procedures address space, and from there to the DB2 system. If the stored procedure definition contains COMMIT ON RETURN NO, DB2 does not commit or roll back any changes from the SQL in the stored procedure until the calling program executes an explicit COMMIT or ROLLBACK statement. If the stored procedure definition contains COMMIT ON RETURN YES, and the stored procedure executed successfully, DB2 commits all changes.

8. Control returns to the calling application, which receives the output parameters and the result set. DB2 then:
   - Closes all cursors that the stored procedure opened, except those that the stored procedure opened to return result sets.
   - Discards all SQL statements that the stored procedure prepared.
   - Reclaims the working storage that the stored procedure used.

   The application can call more stored procedures, or it can execute more SQL statements. DB2 receives and processes the COMMIT or ROLLBACK request. The COMMIT or ROLLBACK operation covers all SQL operations, whether executed by the application or by stored procedures, for that unit of work.

   If the application involves IMS or CICS, similar processing occurs based on the IMS or CICS sync point rather than on an SQL COMMIT or ROLLBACK statement.

9. DB2 returns a reply message to the application describing the outcome of the COMMIT or ROLLBACK operation.

10. The workstation application executes the following steps to retrieve the contents of table EMPPROJACT, which the stored procedure has returned in a result set:
    a. Declares a result set locator for the result set being returned.
    b. Executes the ASSOCIATE LOCATORS statement to associate the result set locator with the result set.
    c. Executes the ALLOCATE CURSOR statement to associate a cursor with the result set.
    d. Executes the FETCH statement with the allocated cursor multiple times to retrieve the rows in the result set.

Setting up the stored procedures environment

This section discusses the tasks that must be performed before stored procedures can run. Most of this information is for system administrators, but application programmers should read "Defining your stored procedure to DB2" on page 531. That section explains how to use the CREATE PROCEDURE statement to define a stored procedure to DB2.

Perform these tasks to prepare the DB2 subsystem to run stored procedures:

- Decide whether to use WLM-established address spaces or DB2-established address spaces for stored procedures.

  See Part 5 (Volume 2) of *DB2 Administration Guide* for a comparison of the two environments.

  If you are currently using DB2-established address spaces and want to convert to WLM-established address spaces, see "Moving stored procedures to a WLM-established environment (for system administrators)" on page 536 for information on what you need to do.

- Define JCL procedures for the stored procedures address spaces

  Member DSNTIJMV of data set DSN710.SDSNSAMP contains sample JCL procedures for starting WLM-established and DB2-established address spaces. If you enter a WLM procedure name or a DB2 procedure name in installation panel DSNTIPX, DB2 customizes a JCL procedure for you. See Part 2 of *DB2 Installation Guide* for details.

- For WLM-established address spaces, define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.

  See Part 5 (Volume 2) of *DB2 Administration Guide* for information on how to do this.

- If you plan to execute stored procedures that use the ODBA interface to access IMS databases, modify the startup procedures for the address spaces in which those stored procedures will run in the following way:

  - Add the data set name of the IMS data set that contains the ODBA callable interface code (usually IMS.RESLIB) to the end of the STEPLIB concatenation.

  - After the STEPLIB DD statement, add a DFSRESLB DD statement that names the IMS data set that contains the ODBA callable interface code.

- Install Language Environment and the appropriate compilers.

  See *OS/390 Language Environment for OS/390 & VM Customization* for information on installing Language Environment.

  See "Language requirements for the stored procedure and its caller" on page 538 for minimum compiler and Language Environment requirements

Perform these tasks for each stored procedure:

- Be sure that the library in which the stored procedure resides is the STEPLIB concatenation of the startup procedure for the stored procedures address space.

- Use the CREATE PROCEDURE statement to define the stored procedure to DB2 and ALTER PROCEDURE to modify the definition.

  See "Defining your stored procedure to DB2" on page 531 for details.

- Perform security tasks for the stored procedure.

  See Part 3 of *DB2 Administration Guide* for more information.

Defining your stored procedure to DB2

Before a stored procedure can run, you must define it to DB2. Use the SQL statement CREATE PROCEDURE to define a stored procedure to DB2. To alter the definition, use the ALTER PROCEDURE statement.

Table 58 lists the characteristics of a stored procedure and the CREATE PROCEDURE and ALTER PROCEDURE parameters that correspond to those characteristics.

Table 58. Characteristics of a stored procedure

Characteristic	CREATE/ALTER PROCEDURE parameter
Stored procedure name Parameter declarations	PROCEDURE
Parameter types and encoding schemes	PROCEDURE
External name	EXTERNAL NAME
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE COMPJAVA LANGUAGE JAVA LANGUAGE PLI LANGUAGE REXX LANGUAGE SQL
Deterministic or not deterministic	NOT DETERMINISTIC DETERMINISTIC
Types of SQL statements in the stored procedure	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA
Parameter style	PARAMETER STYLE DB2SQL[1] PARAMETER STYLE GENERAL PARAMETER STYLE GENERAL WITH NULLS PARAMETER STYLE JAVA
Address space for stored procedures	FENCED
Package Collection	NO COLLID COLLID *collection-id*
WLM Environment	WLM ENVIRONMENT *name* WLM ENVIRONMENT *name,** NO WLM ENVIRONMENT[1]
How long a stored procedure can run	ASUTIME NO LIMIT ASUTIME LIMIT *integer*
Load module stays in memory	STAY RESIDENT NO STAY RESIDENT YES
Program type	PROGRAM TYPE MAIN[1] PROGRAM TYPE SUB
Security	SECURITY DB2 SECURITY USER SECURITY DEFINER
Run-time options	RUN OPTIONS *options*[2]
Maximum number of result sets returned	DYNAMIC RESULT SETS *integer*
Commit work on return from stored procedure	COMMIT ON RETURN YES COMMIT ON RETURN NO

Table 58. Characteristics of a stored procedure (continued)

Characteristic	CREATE/ALTER PROCEDURE parameter
Call with null arguments	CALLED ON NULL INPUT
Pass DB2 environment information	NO DBINFO DBINFO³
Encoding scheme for all string parameters	PARAMETER CCSID EBCDIC PARAMETER CCSID ASCII PARAMETER CCSID UNICODE

Notes:

1. This value is invalid for a REXX stored procedure.

2. This value is ignored for a REXX stored procedure.

3. DBINFO is valid only with PARAMETER STYLE DB2SQL.

For a complete explanation of the parameters in a CREATE PROCEDURE or ALTER PROCEDURE statement, see Chapter 5 of *DB2 SQL Reference*.

For information on the parameters for the CREATE PROCEDURE or ALTER PROCEDURE statement, see Chapter 5 of *DB2 SQL Reference*.

Passing environment information to the stored procedure

If you specify the DBINFO parameter when you define a stored procedure with PARAMETER STYLE DB2SQL, DB2 passes a structure to the stored procedure that contains environment information. Because the structure is also used for user-defined functions, some fields in the structure are not used for stored procedures. The DBINFO structure includes the following information:

Location name length
An unsigned 2-byte integer field. It contains the length of the location name in the next field.

Location name
A 128-byte character field. It contains the name of the location to which the invoker is currently connected.

Authorization ID length
An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

Authorization ID
A 128-byte character field. It contains the authorization ID of the application from which the stored procedure is invoked, padded on the right with blanks. If this stored procedure is nested within other routines (user-defined functions or stored procedures), this value is the authorization ID of the application that invoked the highest-level routine.

Subsystem code page
A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs and encoding scheme of the subsystem from which the user-defined function is invoked. The first nine fields are arranged in an array of three inner structures, each of which contains three integer fields. The three fields in each inner structure contain an SBCS, a DBCS, and a mixed CCSID. The first of the three inner structures is for EBCDIC CCSIDs. The second inner structure is for ASCII CCSIDs. The third inner structure is for Unicode CCSIDs. The last integer field in the outer structure is an index into the array of inner structures.

Table qualifier length
An unsigned two-byte integer field. This field contains 0.

Table qualifier
A 128-byte character field. This field is not used for stored procedures.

Table name length
An unsigned 2-byte integer field. This field contains 0.

Table name
A 128-byte character field. This field is not used for stored procedures.

Column name length
An unsigned 2-byte integer field. This filed contains 0.

Column name
A 128-byte character field. This field is not used for stored procedures.

Product information
An 8-byte character field that identifies the product on which the stored procedure executes. This field has the form *pppvvrrm*, where:

- *ppp* is a 3-byte product code:

  DSN DB2 for OS/390 and z/OS

  ARI DB2 Server for VSE & VM

  QSQ DB2 for AS/400

  SQL DB2 UDB
- *vv* is a 2-digit version identifier.
- *rr* is a 2-digit release identifier.
- *m* is a 1-digit modification level identifier.

Operating system
A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function runs. The value is one of these:

0   Unknown

1   OS/2

3   Windows

4   AIX

5   Windows NT

6   HP-UX

7   Solaris

8   OS/390

13 Siemens Nixdorf

15 Windows 95

16 SCO Unix

Number of entries in table function column list
An unsigned 2-byte integer field. This field contains 0.

Reserved area
24 bytes.

Table function column list pointer
This field is not used for stored procedures.

Unique application identifier
This field is a pointer to a string that uniquely identifies the application's connection to DB2. The string is regenerated at for each connection to DB2.

The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a 1- to 8-character network ID, a period, and a 1- to 8-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

Reserved area
20 bytes.

Example of a stored procedure definition

Suppose you have written and prepared a stored procedure that has these characteristics:

- The name is B.
- It takes two parameters:
  - An integer input parameter named V1
  - A character output parameter of length 9 named V2
- It is written in the C language.
- It contains no SQL statements.
- The same input always produces the same output.
- The load module name is SUMMOD.
- The package collection name is SUMCOLL.
- It should run for no more than 900 CPU service units.
- The parameters can have null values.
- It should be deleted from memory when it completes.
- The Language Environment run-time options it needs are:

  `MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)`

- It is part of the WLM application environment named PAYROLL.
- It runs as a main program.
- It does not access non-DB2 resources, so it does not need a special RACF environment.
- It can return at most 10 result sets.
- When control returns to the client program, DB2 should not commit updates automatically.

This CREATE PROCEDURE statement defines the stored procedure to DB2:

```
CREATE PROCEDURE B(V1 INTEGER IN, CHAR(9) OUT)
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  EXTERNAL NAME SUMMOD
  COLLID SUMCOLL
  ASUTIME LIMIT 900
  PARAMETER STYLE GENERAL WITH NULLS
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
```

```
WLM ENVIRONMENT PAYROLL
PROGRAM TYPE MAIN
SECURITY DB2
DYNAMIC RESULT SETS 10
COMMIT ON RETURN NO;
```

Later, you need to make the following changes to the stored procedure definition:

- It selects data from DB2 tables but does not modify DB2 data.
- The parameters can have null values, and the stored procedure can return a diagnostic string.
- The length of time the stored procedure runs should not be limited.
- If the stored procedure is called by another stored procedure or a user-defined function, the stored procedure uses the WLM environment of the caller.

Execute this ALTER PROCEDURE statement to make the changes:

```
ALTER PROCEDURE B
  READS SQL DATA
  ASUTIME NO LIMIT
  PARAMETER STYLE DB2SQL
  WLM ENVIRONMENT (PAYROLL,*);
```

Refreshing the stored procedures environment (for system administrators)

Depending on what has changed in a stored procedures environment, you might need to perform one or more of these tasks:

- Refresh Language Environment.

  Do this when someone has modified a load module for a stored procedure, and that load module is cached in a stored procedures address space. When you refresh Language Environment, the cached load module is purged. On the next invocation of the stored procedure, the new load module is loaded.

- Restart a stored procedures address space.

  You might stop and then start a stored procedures address space because you need to make a change to the startup JCL for a stored procedures address space.

The method that you use to perform these tasks depends on whether you are using WLM-established or DB2-established address spaces.

For DB2-established address spaces: Use the DB2 commands START PROCEDURE and STOP PROCEDURE to perform all of these tasks.

For WLM-established address spaces:

- If WLM is operating in goal mode:
  - Use this MVS command to refresh a WLM environment when you need to load a new version of a stored procedure. Refreshing the WLM environment refreshes Language Environment.

    ```
    VARY WLM,APPLENV=name,REFRESH
    ```

    name is the name of a WLM application environment associated with a group of stored procedures. This means that when you execute this command, you affect all stored procedures associated with the application environment.

You can call the DB2-supplied stored procedure WLM_REFRESH to refresh a WLM environment from a remote workstation. For information on WLM_REFRESH, see "The WLM environment refresh stored procedure (WLM_REFRESH)" on page 931.

- Use this MVS command to stop all stored procedures address spaces associated with WLM application environment *name*.

```
VARY WLM,APPLENV=name,QUIESCE
```

- Use this MVS command to start all stored procedures address spaces associated with WLM application environment *name*.

```
VARY WLM,APPLENV=name,RESUME
```

See *OS/390 MVS Planning: Workload Management* for more information on the command VARY WLM.

- If WLM is operating in compatibility mode:
  - Use this MVS command to stop a WLM-established stored procedures address space.

```
CANCEL address-space-name
```

  - Use this MVS command to start a WLM-established stored procedures address space.

```
START address-space-name
```

In compatibility mode, you must stop and start stored procedures address spaces when you refresh Language Environment.

Moving stored procedures to a WLM-established environment (for system administrators)

If your DB2 subsystem is installed on OS/390 Release 3 or a subsequent release, you can run some or all or your stored procedures in WLM-established address spaces. To move stored procedures from a DB2-established environment to a WLM-established environment, follow these steps:

1. Define JCL procedures for the stored procedures address spaces

   Member DSNTIJMV of data set DSN710.SDSNSAMP contains sample JCL procedures for starting WLM-established address spaces.

2. Define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.

   See Part 5 (Volume 2) of *DB2 Administration Guide* for information on how to do this.

3. Enter the DB2 command STOP PROCEDURE(*) to stop all activity in the DB2-established stored procedures address space.

4. For each stored procedure, execute ALTER PROCEDURE with the WLM ENVIRONMENT parameter to specify the name of the application environment.

5. Relink all of your existing stored procedures with DSNRLI, the language interface module for the Recoverable Resource Manager Services attachment facility (RRSAF). Use JCL and linkage editor control statements similar to those shown in Figure 162 on page 537.

```
//LINKRRS  EXEC PGM=IEWL,
//     PARM='LIST,XREF,RENT,AMODE=31,RMODE=ANY'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DISP=SHR,DSN=USER.RUNLIB.LOAD
//         DD DISP=SHR,DSN=DSN710.SDSNLOAD
//SYSLMOD  DD DISP=SHR,DSN=USER.RUNLIB.LOAD
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN   DD DISP=SHR,DSN=DSN710.SDSNLOAD
 ENTRY STORPROC
 REPLACE DSNALI(DSNRLI)
 INCLUDE SYSLMOD(STORPROC)
 NAME STORPROC(R)
```

Figure 162. Linking existing stored procedures with RRSAF

6. If WLM is operating in compatibility mode, use the MVS command

   START *address-space-name*

   to start the new WLM-established stored procedures address spaces.

   If WLM is operating in goal mode, the address spaces start automatically.

Redefining stored procedures defined in SYSIBM.SYSPROCEDURES

Before DB2 Version 6, stored procedures were defined to DB2 by inserting a rows into catalog table SYSIBM.SYSPROCEDURES. When you migrate to DB2 Version 6, DB2 automatically creates new definitions for your old stored procedures in SYSIBM.SYSROUTINES and definitions for the stored procedure parameters in SYSIBM.SYSPARMS. However, if you specified values for AUTHID or LUNAME in any old stored procedure definitions, DB2 cannot create new definitions for those stored procedures, and you must manually redefine those stored procedures.

To check for stored procedures with nonblank AUTHID or LUNAME values, execute this query:

```
SELECT * FROM SYSIBM.SYSPROCEDURES
  WHERE AUTHID<>' ' OR LUNAME<>' ';
```

Then use CREATE PROCEDURE to create definitions for all stored procedures that are identified by the SELECT statement. You cannot specify AUTHID or LUNAME using CREATE PROCEDURE. However, AUTHID and LUNAME let you define several versions of a stored procedure, such as a test version and a production version. You can accomplish the same task by specifying a unique schema name for each stored procedure with the same name. For example, for stored procedure INVENTORY, you might define TEST.INVENTORY and PRODTN.INVENTORY.

Writing and preparing an external stored procedure

A stored procedure is a DB2 application program that runs in a stored procedures address space.

There are two types of stored procedures: *external* stored procedures and *SQL procedures.*. External stored procedures are written in a host language. The source code for an external stored procedure is separate from the definition for the stored procedure. An external stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and DB2 commands issued through IFI. SQL procedures are written using SQL procedures statements, which are part of a CREATE PROCEDURE statement. This section discusses writing and

preparing external stored procedures. "Writing and preparing an SQL procedure" on page 552 discusses writing and preparing SQL procedures.

Language requirements for the stored procedure and its caller

You can write an external stored procedure in Assembler, C, C++, COBOL, Java, REXX or PL/I. All programs must be designed to run using Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions. See "Considerations for C++" on page 140 and "Considerations for object-oriented extensions in COBOL" on page 163 for information on including object-oriented extensions in SQL applications. For a list of the minimum compiler and Language Environment requirements, see *DB2 Release Planning Guide*. For information on writing Java stored procedures, see *DB2 Application Programming Guide and Reference for Java*. For information on writing REXX stored procedures, see "Writing a REXX stored procedure" on page 549.

The program that calls the stored procedure can be in any language that supports the SQL CALL statement. ODBC applications can use an escape clause to pass a stored procedure call to DB2.

Calling other programs

A stored procedure can consist of more than one program, each with its own package. Your stored procedure can call other programs, stored procedures, or user-defined functions. Use the facilities of your programming language to call other programs.

If the stored procedure calls other programs that contain SQL statements, each of those called programs must have a DB2 package. The owner of the package or plan that contains the CALL statement must have EXECUTE authority for all packages that the other programs use.

When a stored procedure calls another program, DB2 determines which collection the called program's package belongs to in one of the following ways:
- If the stored procedure executes SET CURRENT PACKAGESET, the called program's package comes from the collection specified in SET CURRENT PACKAGESET.
- If the stored procedure does not execute SET CURRENT PACKAGESET,
  - If the stored procedure definition contains NO COLLID, DB2 uses the collection ID of the package that contains the SQL statement CALL.
  - If the stored procedure definition contains COLLID *collection-id*, DB2 uses *collection-id*.

When control returns from the stored procedure, DB2 restores the value of the special register CURRENT PACKAGESET to the value it contained before the client program executed the SQL statement CALL.

Using reentrant code

Whenever possible, prepare your stored procedures to be reentrant. Using reentrant stored procedures can lead to improved performance for the following reasons:
- A reentrant stored procedure does not have to be loaded into storage every time it is called.
- A single copy of the stored procedure can be shared by multiple tasks in the stored procedures address space. This decreases the amount of virtual storage used for code in the stored procedures address space.

To prepare a stored procedure as reentrant, compile it as reentrant and link-edit it as reentrant and reusable.

For instructions on compiling programs to be reentrant, see the appropriate language manual. For information on using the binder to produce reentrant and reusable load modules, see *DFSMS/MVS®: Program Management*.

To make a reentrant stored procedure remain resident in storage, specify STAY RESIDENT YES in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

If your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time. A non-reentrant stored procedure must not remain in storage. You therefore need to specify STAY RESIDENT NO in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

Writing a stored procedure as a main program or subprogram

A stored procedure that runs in a WLM-established address space and uses Language Environment Release 1.7 or a subsequent release can be either a main program or a subprogram. A stored procedure that runs as a subprogram can perform better because Language Environment does less processing for it.

In general, a subprogram must do the following extra tasks that Language Environment performs for a main program:
- Initialization and cleanup processing
- Allocating and freeing storage
- Closing all open files before exiting

When you code stored procedures as subprograms, follow these rules:
- Follow the language rules for a subprogram. For example, you cannot perform I/O operations in a PL/I subprogram.
- Avoid using statements that terminate the Language Environment enclave when the program ends. Examples of such statements are STOP or EXIT in a PL/I subprogram, or STOP RUN in a COBOL subprogram. If the enclave terminates when a stored procedure ends, and the client program calls another stored procedure that runs as a subprogram, then Language Environment must build a new enclave. As a result, the benefits of coding a stored procedure as a subprogram are lost.

Table 59 summarizes the characteristics that define a main program and a subprogram.

Table 59. Characteristics of main programs and subprograms

Language	Main program	Subprogram
Assembler	MAIN=YES is specified in the invocation of the CEEENTRY macro.	MAIN=NO is specified in the invocation of the CEEENTRY macro.
C	Contains a main() function. Pass parameters to it through argc and argv.	A fetchable function. Pass parameters to it explicitly.

Table 59. Characteristics of main programs and subprograms (continued)

Language	Main program	Subprogram
COBOL	A COBOL program that does not end with GOBACK	A dynamically loaded subprogram that ends with GOBACK
PL/I	Contains a procedure declared with OPTIONS(MAIN)	A procedure declared with OPTIONS(FETCHABLE)

Figure 163 shows an example of coding a C stored procedure as a subprogram.

```
/****************************************************************/
/*  This C subprogram is a stored procedure that uses linkage   */
/*  convention GENERAL and receives 3 parameters.               */
/****************************************************************/
#pragma linkage(cfunc,fetchable)
#include <stdlib.h>
void cfunc(char p1[11],long *p2,short *p3)
{
  /****************************************************************/
  /*  Declare variables used for SQL operations.  These variables */
  /*  are local to the subprogram and must be copied to and from  */
  /*  the parameter list for the stored procedure call.           */
  /****************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
  EXEC SQL END DECLARE SECTION;
```

Figure 163. A C stored procedure coded as a subprogram (Part 1 of 2)

```
  /**************************************************************/
  /*  Receive input parameter values into local variables.    */
  /**************************************************************/
  strcpy(parm1,p1);
  parm2 = *p2;
  parm3 = *p3;
  /**************************************************************/
  /*  Perform operations on local variables.                  */
  /**************************************************************/

  .
  .
  .
  /**************************************************************/
  /*  Set values to be passed back to the caller.             */
  /**************************************************************/
  strcpy(parm1,"SETBYSP");
  parm2 = 100;
  parm3 = 200;
  /**************************************************************/
  /*  Copy values to output parameters.                       */
  /**************************************************************/
  strcpy(p1,parm1);
  *p2 = parm2;
  *p3 = parm3;
}
```

Figure 163. A C stored procedure coded as a subprogram (Part 2 of 2)

Figure 164 on page 541 shows an example of coding a C++ stored procedure as a subprogram.

```
/******************************************************************/
/*  This C++ subprogram is a stored procedure that uses linkage   */
/*  convention GENERAL and receives 3 parameters.                 */
/*  The extern statement is required.                             */
/******************************************************************/
extern  "C"  void cppfunc(char p1[11],long *p2,short *p3);
#pragma linkage(cppfunc,fetchable)
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;
void cppfunc(char p1[11],long *p2,short *p3)
{
  /******************************************************************/
  /*  Declare variables used for SQL operations.  These variables */
  /*  are local to the subprogram and must be copied to and from  */
  /*  the parameter list for the stored procedure call.           */
  /******************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
  EXEC SQL END DECLARE SECTION;
```

Figure 164. A C++ stored procedure coded as a subprogram (Part 1 of 2)

```
  /************************************************************/
  /*  Receive input parameter values into local variables.   */
  /************************************************************/
  strcpy(parm1,p1);
  parm2 = *p2;
  parm3 = *p3;
  /************************************************************/
  /*  Perform operations on local variables.                 */
  /************************************************************/

  ⋮

  /************************************************************/
  /*  Set values to be passed back to the caller.            */
  /************************************************************/
  strcpy(parm1,"SETBYSP");
  parm2 = 100;
  parm3 = 200;
  /************************************************************/
  /*  Copy values to output parameters.                      */
  /************************************************************/
  strcpy(p1,parm1);
  *p2 = parm2;
  *p3 = parm3;
}
```

Figure 164. A C++ stored procedure coded as a subprogram (Part 2 of 2)

A stored procedure that runs in a DB2-established address space must contain a main program.

Restrictions on a stored procedure

- Do not include explicit attachment facility calls in a stored procedure. Stored procedures running in a DB2-established address space use call attachment facility (CAF) calls implicitly. Stored procedures running in a WLM-established

address space use Recoverable Resource Manager Services attachment facility (RRSAF) calls implicitly. If a stored procedure makes an explicit attachment facility call, DB2 rejects the call.

- Do not include the SET CURRENT SQLID statement in your stored procedure.

  When DB2 encounters this statement, it places the DB2 thread in a *must roll back* state. When control returns to the calling program, the calling program must do one of the following things:

  - Execute the ROLLBACK statement, so that it is free to execute other SQL statements after rollback is complete.
  - Terminate, causing an automatic rollback of the unit of work.

Using COMMIT and ROLLBACK statements in a stored procedure

When you execute COMMIT or ROLLBACK statements in your stored procedure, DB2 commits or rolls back all changes within the unit of work. These changes include changes that the client application made before it called the stored procedure, as well as DB2 work that the stored procedure does.

A stored procedure that includes COMMIT or ROLLBACK statements must be defined with the CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA clause. There is no interaction between the COMMIT ON RETURN clause in a stored procedure definition and COMMIT or ROLLBACK statements in the stored procedure code. If you specify COMMIT ON RETURN YES when you define the stored procedure, DB2 issues a COMMIT when control returns from the stored procedure. This occurs regardless of whether the stored procedure contains COMMIT or ROLLBACK statements.

A ROLLBACK statement has the same effect on cursors in a stored procedure as it has on cursors in stand-alone programs. A ROLLBACK statement closes all open cursors. A COMMIT statement in a stored procedure closes cursors that are not declared WITH HOLD, and leaves cursors open that are declared WITH HOLD. The effect of COMMIT or ROLLBACK on cursors applies to cursors that are declared in the calling application, as well as cursors that are declared in the stored procedure.

Under the following conditions, you *cannot* include COMMIT or ROLLBACK statements in a stored procedure:

- The stored procedure is nested within a trigger or a user-defined function.
- The stored procedure is called by a client that uses two-phase commit processing.
- The client program uses a type 2 connection to connect to the remote server that contains the stored procedure.

You *cannot* include ROLLBACK statements in a stored procedure if DB2 is not the commit coordinator.

If a COMMIT or ROLLBACK statement in a stored procedure violates any of the previous conditions, DB2 puts the transaction in a must-rollback statement, and the CALL statement returns a −751 SQLCODE.

Using special registers in a stored procedure

You can use all special registers in a stored procedure. However, you can modify only some of those special registers. After a stored procedure completes, DB2 restores all special registers to the values they had before invocation.

Table 60 shows information you need to use special registers in a stored procedure.

Table 60. Characteristics of special registers in a stored procedure

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET statement to modify?
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the stored procedure package[1]	The value of bind option ENCODING for the stored procedure package[1]	Yes
CURRENT DATE	New value for each SQL statement in the stored procedure package[2]	New value for each SQL statement in the stored procedure package[2]	Not applicable[5]
CURRENT DEGREE	Inherited from invoker[3]	The value of field CURRENT DEGREE on installation panel DSNTIP4	Yes
CURRENT LOCALE LC_CTYPE	Inherited from invoker	The value of field CURRENT DEGREE on installation panel DSNTIP4	Yes
CURRENT MEMBER	New value for each SET *host-variable*=CURRENT MEMBER statement	New value for each SET *host-variable*=CURRENT MEMBER statement	No
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the stored procedure package or inherited from invoker[6]	The value of bind option OPTHINT for the stored procedure package	Yes
CURRENT PACKAGESET	Inherited from invoker[4]	Inherited from invoker[4]	Yes
CURRENT PATH	The value of bind option PATH for the stored procedure package or inherited from invoker[6]	The value of bind option PATH for the stored procedure package	Yes
CURRENT PRECISION	Inherited from invoker	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT RULES	Inherited from invoker	The value of bind option SQLRULES for the stored procedure package	Yes
CURRENT SERVER	Inherited from invoker	Inherited from invoker	Yes
CURRENT SQLID	The primary authorization ID of the application process or inherited from invoker[7]	The primary authorization ID of the application process	Yes[8]
CURRENT TIME	New value for each SQL statement in the stored procedure package[2]	New value for each SQL statement in the stored procedure package[2]	Not applicable[5]

Table 60. Characteristics of special registers in a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET statement to modify?
CURRENT TIMESTAMP	New value for each SQL statement in the stored procedure package[2]	New value for each SQL statement in the stored procedure package[2]	Not applicable[5]
CURRENT TIMEZONE	Inherited from invoker	Inherited from invoker	Not applicable[5]
CURRENT USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable[5]

Notes:

1. If the ENCODING bind option is not specified, the initial value is the value that was specified in field APPLICATION ENCODING of installation panel DSNTIPF.

2. If the stored procedure is invoked within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the function package.

3. DB2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, DB2 ignores the CURRENT DEGREE value.

4. If the stored procedure definer specifies a value for COLLID in the CREATE FUNCTION statement, DB2 sets CURRENT PACKAGESET to the value of COLLID.

5. Not applicable because no SET statement exists for the special register.

6. If a program within the scope of the invoking program issues a SET statement for the special register before the stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the stored procedure package.

7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.

8. If the stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed but does not affect the authorization ID that is used for the dynamic SQL statements in the stored procedure package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements. See "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 415 for more information on DYNAMICRULES values and authorization IDs.

Accessing other sites in a stored procedure

Stored procedures can access tables at other DB2 locations using 3-part object names or CONNECT statements. If you use CONNECT statements, you use DRDA access to access tables. If you use 3-part object names or aliases for 3-part object names, the distributed access method depends on the value of DBPROTOCOL you specified when you bound the stored procedure package. If you did not specify the DBPROTOCOL bind parameter, the distributed access method depends on the value of field DATABASE PROTOCOL on installation panel DSNTIP5. A value of PRIVATE tells DB2 to use DB2 private protocol access to access remote data for the stored procedure. DRDA tells DB2 to use DRDA access.

When a local DB2 application calls a stored procedure, the stored procedure cannot have DB2 private protocol access to any DB2 sites already connected to the calling program by DRDA access.

The local DB2 application cannot use DRDA access to connect to any location that the stored procedure has already accessed using DB2 private protocol access. Before making the DB2 private protocol connection, the local DB2 application must first execute the RELEASE statement to terminate the DB2 private protocol connection, and then commit the unit of work.

Writing a stored procedure to access IMS databases

IMS Open Database Access (ODBA) support lets a DB2 stored procedure connect to an IMS DBCTL or IMS DB/DC system and issue DL/I calls to access IMS databases.

ODBA support uses OS/390 RRS for syncpoint control of DB2 and IMS resources. Therefore, stored procedures that use ODBA can run only in WLM-established stored procedures address spaces.

When you write a stored procedure that uses ODBA, follow the rules for writing an IMS application program that issues DL/I calls. See *IMS Application Programming: Database Manager* and *IMS Application Programming: Transaction Manager* for information on writing DL/I applications.

IMS work that is performed in a stored procedure is in the same commit scope as the stored procedure. As with any other stored procedure, the calling application commits work.

A stored procedure that uses ODBA must issue a DPSB PREP call to deallocate a PSB when all IMS work under that PSB is complete. The PREP keyword tells IMS to move inflight work to an indoubt state. When work is in the indoubt state, IMS does not require activation of syncpoint processing when the DPSB call is executed. IMS commits or backs out the work as part of RRS two-phase commit when the stored procedure caller executes COMMIT or ROLLBACK.

A sample COBOL stored procedure and client program demonstrate accessing IMS data using the ODBA interface. The stored procedure source code is in member DSN8EC1 and is prepared by job DSNTEJ61. The calling program source code is in member DSN8EC1 and is prepared and executed by job DSNTEJ62. All code is in data set DSN710.SDSNSAMP.

The startup procedure for a stored procedures address space in which stored procedures that use ODBA run must include a DFSRESLB DD statement and an extra data set in the STEPLIB concatenation. See "Setting up the stored procedures environment" on page 530 for more information.

Writing a stored procedure to return result sets to a DRDA client

Your stored procedure can return multiple query result sets to a DRDA client if the following conditions are satisfied:
- The client supports the DRDA code points used to return query result sets.
- The value of DYNAMIC RESULT SETS in the stored procedure definition is greater than 0.

For each result set you want returned, your stored procedure must:

- Declare a cursor with the option WITH RETURN.
- Open the cursor.
- If the cursor is scrollable, ensure that the cursor is positioned before the first row of the result table.
- Leave the cursor open.

When the stored procedure ends, DB2 returns the rows in the query result set to the client.

DB2 does not return result sets for cursors that are closed before the stored procedure terminates. The stored procedure must execute a CLOSE statement for each cursor associated with a result set that should not be returned to the DRDA client.

Example: Suppose you want to return a result set that contains entries for all employees in department D11. First, declare a cursor that describes this subset of employees:

```
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
  SELECT * FROM DSN8710.EMP
    WHERE WORKDEPT='D11';
```

Then, open the cursor:

```
EXEC SQL OPEN C1;
```

DB2 returns the result set and the name of the SQL cursor for the stored procedure to the client.

Use meaningful cursor names for returning result sets: The name of the cursor that is used to return result sets is made available to the client application through extensions to the DESCRIBE statement. See "Writing a DB2 for OS/390 and z/OS client program or SQL procedure to receive result sets" on page 599 for more information.

Use cursor names that are meaningful to the DRDA client application, especially when the stored procedure returns multiple result sets.

Objects from which you can return result sets: You can use any of these objects in the SELECT statement associated with the cursor for a result set:

- Tables, synonyms, views, created temporary tables, declared temporary tables, and aliases defined at the local DB2 system
- Tables, synonyms, views, created temporary tables, and aliases defined at remote DB2 for OS/390 and z/OS systems that are accessible through DB2 private protocol access

Returning a subset of rows to the client: If you execute FETCH statements with a result set cursor, DB2 does not return the fetched rows to the client program. For example, if you declare a cursor WITH RETURN and then execute the statements OPEN, FETCH, FETCH, the client receives data beginning with the third row in the result set. If the result set cursor is scrollable and you fetch rows with it, you need to position the cursor before the first row of the result table after you fetch the rows and before the stored procedure ends.

Using a temporary table to return result sets: You can use a created temporary table or declared temporary table to return result sets from a stored procedure. This capability can be used to return nonrelational data to a DRDA client.

For example, you can access IMS data from a stored procedure in the following way:

- Use MVS/APPC to issue an IMS transaction.
- Receive the IMS reply message, which contains data that should be returned to the client.
- Insert the data from the reply message into a temporary table.
- Open a cursor against the temporary table. When the stored procedure ends, the rows from the temporary table are returned to the client.

Preparing a stored procedure

There are a number of tasks that must be completed before a stored procedure can run on an MVS server. You share these tasks with your system administrator. Part 2 of *DB2 Installation Guide* and "Defining your stored procedure to DB2" on page 531 describe what the system administrator needs to do.

Complete the following steps:

1. Precompile and compile the application.

   If your stored procedure is a COBOL program, you must compile it with the option NODYNAM.

2. Link-edit the application. Your stored procedure must either link-edit or load one of these language interface modules:

   DSNALI
   > The language interface module for the call attachment facility. Link-edit or load this module if your stored procedure runs in a DB2-established address space. For more information, see "Accessing the CAF language interface" on page 735.

   DSNRLI
   > The language interface module for the Recoverable Resource Manager Services attachment facility. Link-edit or load this module if your stored procedure runs in a WLM-established address space. If the stored procedure references LOBs or distinct types, you must link-edit or load DSNRLI. For more information, see "Accessing the RRSAF language interface" on page 766.

   If your stored procedure runs in a WLM-established address space, you must specify the parameter AMODE(31) when you link-edit it.

3. Bind the DBRM to DB2 using the command BIND PACKAGE. Stored procedures require only a package at the server. You do not need to bind a plan. For more information, see "Binding the stored procedure" on page 548.

4. Define the stored procedure to DB2.

5. Use GRANT EXECUTE to authorize the appropriate users to use the stored procedure. For example,

   ```
   GRANT EXECUTE ON PROCEDURE SPSCHEMA.STORPRCA TO JONES;
   ```

   That allows an application running under authorization ID JONES to call stored procedure SPSCHEMA.STORPRCA.

Preparing a stored procedure to run as an authorized program: If your stored procedure runs in a WLM-established address space, you can run it as an MVS authorized program. To prepare a stored procedure to run as an authorized program, do these additional things:

- When you link-edit the stored procedure:

– Indicate that the load module can use restricted system services by specifying the parameter value AC=1.
– Put the load module for the stored procedure in an APF-authorized library.
- Be sure that the stored procedure runs in an address space with a startup procedure in which all libraries in the STEPLIB concatenation are APF-authorized. Specify an application environment WLM ENVIRONMENT parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure that ensures that the stored procedure runs in an address space with this characteristic.

Binding the stored procedure

A stored procedure does not require a DB2 plan. A stored procedure runs under the caller's thread, using the plan from the client program that calls it.

The calling application can use a DB2 package or plan to execute the CALL statement. The stored procedure must use a DB2 package as Figure 165 shows.
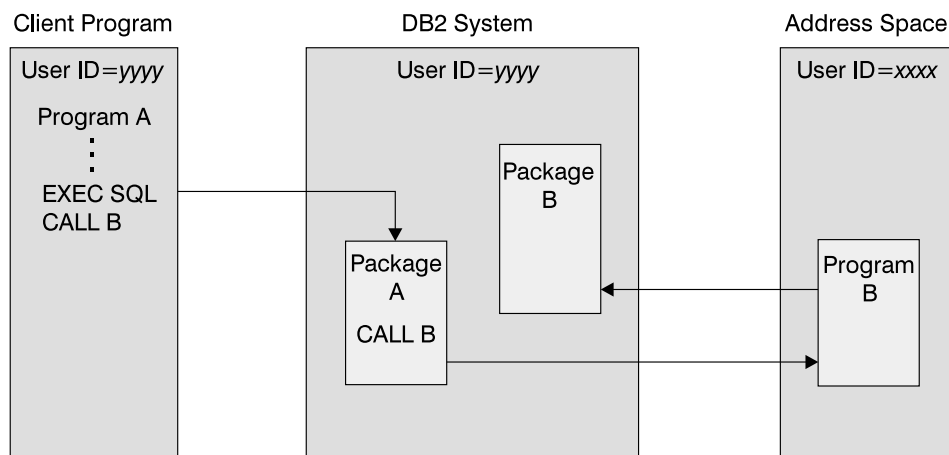
Figure 165. Stored procedures run-time environment

When you bind a stored procedure:
- Use the command BIND PACKAGE to bind the stored procedure. If you use the option ENABLE to control access to a stored procedure package, you must enable the system connection type of the application that executes the CALL statement.
- The package for the stored procedure does not need to be bound with the plan for the program that calls it.
- The owner of the package that contains the SQL statement CALL must have the EXECUTE privilege on all packages that the stored procedure accesses, including packages named in SET CURRENT PACKAGESET.

The following must exist at the server, as shown in Figure 165:
- A plan or package containing the SQL statement CALL. This package is associated with the client program.
- A package associated with the stored procedure.

The server program might use more than one package. These packages come from two sources:

- A DBRM that you bind several times into several versions of the same package, all with the same package name, which can then reside in different collections. Your stored procedure can switch from one version to another by using the statement SET CURRENT PACKAGESET.
- A package associated with another program that contains SQL statements that the stored procedure calls.

Writing a REXX stored procedure

A REXX stored procedure is much like any other REXX procedure and follows the same rules as stored procedures in other languages. It receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. A REXX stored procedure is different from other REXX procedures in the following ways:

- A REXX stored procedure cannot execute the ADDRESS DSNREXX CONNECT and ADDRESS DSNREXX DISCONNECT commands. When you execute SQL statements in your stored procedure, DB2 establishes the connection for you.
- A REXX stored procedure must run in a WLM-established stored procedures address space.
- As in other stored procedures, you cannot include the following statements in a REXX stored procedure:
  - COMMIT
  - SET CURRENT SQLID

Unlike other stored procedures, you do not prepare REXX stored procedures for execution. REXX stored procedures run using one of four packages that are bound during the installation of DB2 REXX Language Support. The package that DB2 uses when the stored procedure runs depends on the current isolation level at which the stored procedure runs:

Package name	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

Figure 167 on page 550 shows an example of a REXX stored procedure that executes DB2 commands. The stored procedure performs the following actions:

- Receives one input parameter, which contains a DB2 command.
- Calls the IFI COMMAND function to execute the command.
- Extracts the command result messages from the IFI return area and places the messages in a created temporary table. Each row of the temporary table contains a sequence number and the text of one message.
- Opens a cursor to return a result set that contains the command result messages.
- Returns the unformatted contents of the IFI return area in an output parameter.

Figure 166 on page 550 shows the definition of the stored procedure.

```
CREATE PROCEDURE COMMAND(IN CMDTEXT VARCHAR(254), OUT CMDRESULT VARCHAR(32704))
  LANGUAGE REXX
  EXTERNAL NAME COMMAND
  NO COLLID
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL
  STAY RESIDENT NO
  RUN OPTIONS 'TRAP(ON)'
  WLM ENVIRONMENT WLMENV1
  SECURITY DB2
  DYNAMIC RESULT SETS 1
  COMMIT ON RETURN NO;
```

Figure 166. Definition for REXX stored procedure COMMAND

```
/* REXX */
PARSE UPPER ARG CMD                        /* Get the DB2 command text */
                                           /* Remove enclosing quotes  */
IF LEFT(CMD,2) = ""'" & RIGHT(CMD,2) = "'"" THEN
CMD = SUBSTR(CMD,2,LENGTH(CMD)-2)
ELSE
IF LEFT(CMD,2) = """'" & RIGHT(CMD,2) = "'""" THEN
CMD = SUBSTR(CMD,3,LENGTH(CMD)-4)
COMMAND    = SUBSTR("COMMAND",1,18," ")
  /*****************************************************************/
  /*  Set up the IFCA, return area, and output area for the       */
  /*  IFI COMMAND call.                                           */
  /*****************************************************************/
IFCA = SUBSTR('00'X,1,180,'00'X)
IFCA = OVERLAY(D2C(LENGTH(IFCA),2),IFCA,1+0)
IFCA = OVERLAY("IFCA",IFCA,4+1)
RTRNAREASIZE = 262144 /*1048572*/
RTRNAREA = D2C(RTRNAREASIZE+4,4)LEFT(' ',RTRNAREASIZE,' ')
OUTPUT = D2C(LENGTH(CMD)+4,2)||'0000'X||CMD
BUFFER = SUBSTR(" ",1,16," ")
  /*****************************************************************/
  /*  Make the IFI COMMAND call.                                  */
  /*****************************************************************/
ADDRESS LINKPGM "DSNWLIR COMMAND IFCA RTRNAREA OUTPUT"
WRC = RC
RTRN=    SUBSTR(IFCA,12+1,4)
REAS=    SUBSTR(IFCA,16+1,4)
TOTLEN = C2D(SUBSTR(IFCA,20+1,4))
  /*****************************************************************/
  /*  Set up the host command environment for SQL calls.         */
  /*****************************************************************/
"SUBCOM DSNREXX"                           /* Host cmd env available? */
IF RC THEN                                 /* No--add host cmd env    */
  S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
```

Figure 167. Example of a REXX stored procedure: COMMAND (Part 1 of 3)

```
      /****************************************************************/
      /*  Set up SQL statements to insert command output messages    */
      /*  into a temporary table.                                    */
      /****************************************************************/
SQLSTMT='INSERT INTO SYSIBM.SYSPRINT(SEQNO,TEXT) VALUES(?,?)'
ADDRESS DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1"
IF SQLCODE ¬= 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL PREPARE S1 FROM :SQLSTMT"
IF SQLCODE ¬= 0 THEN CALL SQLCA
      /****************************************************************/
      /*  Extract messages from the return area and insert them into */
      /*  the temporary table.                                       */
      /****************************************************************/
SEQNO = 0
OFFSET = 4+1
DO WHILE ( OFFSET < TOTLEN )
  LEN = C2D(SUBSTR(RTRNAREA,OFFSET,2))
  SEQNO = SEQNO + 1
  TEXT = SUBSTR(RTRNAREA,OFFSET+4,LEN-4-1)
  ADDRESS DSNREXX "EXECSQL EXECUTE S1 USING :SEQNO,:TEXT"
  IF SQLCODE ¬= 0 THEN CALL SQLCA
  OFFSET = OFFSET + LEN
END
      /****************************************************************/
      /*  Set up a cursor for a result set that contains the command */
      /*  output messages from the temporary table.                  */
      /****************************************************************/
SQLSTMT='SELECT SEQNO,TEXT FROM SYSIBM.SYSPRINT ORDER BY SEQNO'
ADDRESS DSNREXX "EXECSQL DECLARE C2 CURSOR FOR S2"
IF SQLCODE ¬= 0 THEN CALL SQLCA

ADDRESS DSNREXX "EXECSQL PREPARE S2 FROM :SQLSTMT"
IF SQLCODE ¬= 0 THEN CALL SQLCA
      /****************************************************************/
      /*  Open the cursor to return the message output result set to */
      /*  the caller.                                                */
      /****************************************************************/
ADDRESS DSNREXX "EXECSQL OPEN C2"
IF SQLCODE ¬= 0 THEN CALL SQLCA

S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX') /* REMOVE CMD ENV  */

EXIT SUBSTR(RTRNAREA,1,TOTLEN+4)
```

Figure 167. Example of a REXX stored procedure: COMMAND (Part 2 of 3)

```
                   /***************************************************************/
                   /*  Routine to display the SQLCA                              */
                   /***************************************************************/
             SQLCA:
             SAY 'SQLCODE ='SQLCODE
             SAY 'SQLERRMC ='SQLERRMC
             SAY 'SQLERRP ='SQLERRP
             SAY 'SQLERRD ='SQLERRD.1',',
                            || SQLERRD.2',',
                            || SQLERRD.3',',
                            || SQLERRD.4',',
                            || SQLERRD.5',',
                            || SQLERRD.6

             SAY 'SQLWARN ='SQLWARN.0',',
                            || SQLWARN.1',',
                            || SQLWARN.2',',
                            || SQLWARN.3',',
                            || SQLWARN.4',',
                            || SQLWARN.5',',
                            || SQLWARN.6',',
                            || SQLWARN.7',',
                            || SQLWARN.8',',
                            || SQLWARN.9',',
                            || SQLWARN.10
             SAY 'SQLSTATE='SQLSTATE
             SAY 'SQLCODE ='SQLCODE
             EXIT 'SQLERRMC ='SQLERRMC';' ,
               || 'SQLERRP ='SQLERRP';' ,
               || 'SQLERRD ='SQLERRD.1',',
                            || SQLERRD.2',',
                            || SQLERRD.3',',
                            || SQLERRD.4',',
                            || SQLERRD.5',',
                            || SQLERRD.6';' ,

               || 'SQLWARN ='SQLWARN.0',',
                            || SQLWARN.1',',
                            || SQLWARN.2',',
                            || SQLWARN.3',',
                            || SQLWARN.4',',
                            || SQLWARN.5',',
                            || SQLWARN.6',',
                            || SQLWARN.7',',
                            || SQLWARN.8',',
                            || SQLWARN.9',',
                            || SQLWARN.10';' ,
               || 'SQLSTATE='SQLSTATE';'
```

Figure 167. Example of a REXX stored procedure: COMMAND (Part 3 of 3)

Writing and preparing an SQL procedure

An SQL procedure is a stored procedure in which the source code for the procedure is in an SQL CREATE PROCEDURE statement. The part of the CREATE PROCEDURE statement that contains the code is called the *procedure body*.

Creating an SQL procedure involves writing the source statements for the SQL procedure, creating the executable form of the SQL procedure, and defining the SQL procedure to DB2. There are two ways to create an SQL procedure:

- Use the IBM DB2 Stored Procedure Builder product to specify the source statements for the SQL procedure, define the SQL procedure to DB2, and prepare the SQL procedure for execution.
- Write a CREATE PROCEDURE statement for the SQL procedure. Then use one of the methods in "Preparing an SQL procedure" on page 561 to define the SQL procedure to DB2 and create an executable procedure.

This section discusses how to write a and prepare an SQL procedure. The following topics are included:
- "Comparison of an SQL procedure and an external procedure"
- "Statements that you can include in a procedure body" on page 554
- "Terminating statements in an SQL procedure" on page 557
- "Handling errors in an SQL procedure" on page 557
- "Examples of SQL procedures" on page 559
- "Preparing an SQL procedure" on page 561

For information on the syntax of the CREATE PROCEDURE statement and the procedure body, see *DB2 SQL Reference* .

Comparison of an SQL procedure and an external procedure

Like an external stored procedure, an SQL procedure consists of a stored procedure definition and the code for the stored procedure program.

An external stored procedure definition and an SQL procedure definition specify the following common information:
- The procedure name.
- Input and output parameter attributes.
- The language in which the procedure is written. For an SQL procedure, the language is SQL.
- Information that will be used when the procedure is called, such as run-time options, length of time that the procedure can run, and whether the procedure returns result sets.

An external stored procedure and an SQL procedure differ in the way that they specify the code for the stored procedure. An external stored procedure definition specifies the name of the stored procedure program. An SQL procedure definition contains the source code for the stored procedure.

For an external stored procedure, you define the stored procedure to DB2 by executing the CREATE PROCEDURE statement. You change the definition of the stored procedure by executing the ALTER PROCEDURE statement. For an SQL procedure, you define the stored procedure to DB2 by preprocessing a CREATE PROCEDURE statement, then executing the CREATE PROCEDURE statement statically or dynamically. As with an external stored procedure, you change the definition by executing the ALTER PROCEDURE statement. You cannot change the procedure body with the ALTER PROCEDURE statement. See "Preparing an SQL procedure" on page 561 for more information on defining an SQL procedure to DB2.

Figure 168 on page 554 shows a definition for an external stored procedure that is written in COBOL. The stored procedure program, which updates employee salaries, is called UPDSAL.

Figure 169 on page 554 shows a definition for an equivalent SQL procedure.

```
CREATE PROCEDURE UPDATESALARY1        1
 (IN EMPNUMBR CHAR(10),               2
 IN RATE DECIMAL(6,2))
 LANGUAGE COBOL                       3
 EXTERNAL NAME UPDSAL;                4
```

Figure 168. Example of an external stored procedure definition

Notes to Figure 168:

1	The stored procedure name is UPDATESALARY1.
2	The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.
3	LANGUAGE COBOL indicates that this is an external procedure, so the code for the stored procedure is in a separate, COBOL program.
4	The name of the load module that contains the executable stored procedure program is UPDSAL.

```
CREATE PROCEDURE UPDATESALARY1        1
 (IN EMPNUMBR CHAR(10),               2
 IN RATE DECIMAL(6,2))
 LANGUAGE SQL                         3
  UPDATE EMP                          4
    SET SALARY = SALARY * RATE
    WHERE EMPNO = EMPNUMBR
```

Figure 169. Example of an SQL procedure definition

Notes to Figure 169:

1	The stored procedure name is UPDATESALARY1.
2	The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.
3	LANGUAGE SQL indicates that this is an SQL procedure, so a procedure body follows the other parameters.
4	The procedure body consists of a single SQL UPDATE statement, which updates rows in the employee table.

Statements that you can include in a procedure body

A procedure body consists of a single simple or compound statement. The types of statements that you can include in a procedure body are:

Assignment statement
Assigns a value to an output parameter or to an SQL variable, which is a variable that is defined and used only within a procedure body. The right side of an assignment statement can include SQL built-in functions.

CALL statement
Calls another stored procedure. This statement is similar to the CALL statement described in Chapter 5 of *DB2 SQL Reference*, except that the parameters must be SQL variables, parameters for the SQL procedure, or constants.

CASE statement
Selects an execution path based on the evaluation of one or more conditions. This statement is similar to the CASE expression, which is described in Chapter 2 of *DB2 SQL Reference*.

GET DIAGNOSTICS statement
Obtains information about the previous SQL statement that was executed.

GOTO statement
Transfers program control to a labelled statement.

IF statement
Selects an execution path based on the evaluation of a condition.

LEAVE statement
Transfers program control out of a loop or a block of code.

LOOP statement
Executes a statement or group of statements multiple times.

REPEAT statement
Executes a statement or group of statements until a search condition is true.

WHILE statement
Repeats the execution of a statement or group of statements while a specified condition is true.

Compound statement
Can contain one or more of any of the other types of statements in this list. In addition, a compound statement can contain SQL variable declarations, condition handlers, or cursor declarations.

The order of statements in a compound statement must be:

1. SQL variable and condition declarations
2. Cursor declarations
3. Handler declarations
4. Procedure body statements (CALL, CASE, IF, LOOP, REPEAT, WHILE, SQL)

SQL statement
A subset of the SQL statements that are described in Chapter 5 of *DB2 SQL Reference*. Certain SQL statements are valid in a compound statement, but not valid if the SQL statement is the only statement in the procedure body. Appendix B of *DB2 SQL Reference* lists the SQL statements that are valid in an SQL procedure.

See the discussion of the procedure body in *DB2 SQL Reference* for detailed descriptions and syntax of each of these statements.

Declaring and using variables in an SQL procedure

To store data that you use only within an SQL procedure, you can declare *SQL variables.* SQL variables are the equivalent of host variables in external stored procedures. SQL variables can have the same data types and lengths as SQL procedure parameters. For a discussion of data types and lengths, see the CREATE PROCEDURE discussion in Chapter 5 of *DB2 SQL Reference*.

The general form of an SQL variable declaration is:

```
DECLARE SQL-variable-name data-type;
```

The general form of a declaration for an SQL variable that you use as a result set locator is:

```
DECLARE SQL-variable-name data-type RESULT_SET_LOCATOR VARYING;
```

The general form of a declaration for an SQL variable that you use as a table locator is:

```
DECLARE SQL-variable-name data-type TABLE LIKE table-name AS LOCATOR;
```

SQL variables have these restrictions:

- SQL variable names can be up to 64 bytes in length. They can include alphanumeric characters and the underscore character. Condition names and label names also have these restrictions.
- Because DB2 folds all SQL variables to uppercase, you cannot declare two SQL variables that are the same except for case. For example, you cannot declare two SQL variables named varx and VARX.
- If you refer to an SQL procedure parameter in the procedure body, you cannot declare an SQL variable with a name that is the same as that parameter name.
- You cannot use an SQL reserved word as an SQL variable name, even if that SQL reserved word is delimited.
- When you use an SQL variable in an SQL statement, do not precede the variable with a colon.
- When you call a user-defined function from an SQL procedure, and the user-defined function definition includes parameters of type CHAR, you need to cast the corresponding parameter values in the user-defined function invocation to CHAR to ensure that DB2 invokes the correct function. For example, suppose that an SQL procedure calls user-defined function CVRTNUM, which takes one input parameter of type CHAR(6). Also suppose that you declare SQL variable EMPNUMBR in the SQL procedure. When you invoke CVRTNUM, cast EMPNUMBR to CHAR:

```
UPDATE EMP
 SET EMPNO=CVRTNUM(CHAR(EMPNUMBR))
 WHERE EMPNO = EMPNUMBR;
```

- Within a procedure body, the following rules apply to IN, OUT, and INOUT parameters:
  - You can use a parameter that you define as IN on the left or right side of an assignment statement. However, if you assign a value to an IN parameter, you cannot pass the new value back to the caller. The IN parameter has the same value before and after the SQL procedure is called.
  - You can use a parameter that you define as OUT on the left or right side of an assignment statement. The last value that you assign to the parameter is the value that is returned to the caller.
  - You can use a parameter that you define as INOUT on the left or right side of an assignment statement. The caller determines the first value of the INOUT parameter, and the last value that you assign to the parameter is the value that is returned to the caller.

You can perform any operations on SQL variables that you can perform on host variables in SQL statements.

Qualifying SQL variable names and other object names is a good way to avoid ambiguity. Use the following guidelines to determine when to qualify variable names:

- When you use an SQL procedure parameter in the procedure body, qualify the parameter name with the procedure name.
- Specify a label for each compound statement, and qualify SQL variable names in the compound statement with that label.
- Qualify column names with the associated table or view names.

> **Important**
>
> The way that DB2 determines the qualifier for unqualified names might change in the future. To avoid changing your code later, qualify all SQL variable names.

Parameter style for an SQL procedure

DB2 supports only the GENERAL WITH NULLS linkage convention for SQL procedures. This means that when you call an SQL procedure, you must include an indicator variable with each parameter in the CALL statement. See "Linkage conventions" on page 572 for more information on stored procedure linkage conventions.

Terminating statements in an SQL procedure

The way that you terminate a statement in an SQL procedure depends on the use of the statement in that procedure:

- A procedure body has no terminating character. Therefore, if an SQL procedure statement is the outermost of a set of nested statements, or if the statement is the only statement in the procedure body, that statement does not have a terminating character.

- If a statement is nested within other statements in the procedure body, that statement ends with a semicolon.

Handling errors in an SQL procedure

If an SQL error occurs when an SQL procedure executes, the SQL procedure ends unless you include statements called *handlers* to tell the procedure to perform some other action. Handlers are similar to WHENEVER statements in external SQL application programs. Handlers tell the SQL procedure what to do when an SQL error or SQL warning occurs, or when no more rows are returned from a query. In addition, you can declare handlers for specific SQLSTATEs. You can refer to an SQLSTATE by its number in a handler, or you can declare a name for the SQLSTATE, then use that name in the handler.

The general form of a handler declaration is:

```
DECLARE handler-type HANDLER FOR condition SQL-procedure-statement;
```

In general, the way that a handler works is that when an error occurs that matches *condition*, *SQL-procedure-statement* executes. When *SQL-procedure-statement* completes, DB2 performs the action that is indicated by *handler-type*.

There are two types of handlers:

CONTINUE
> Specifies that after *SQL-procedure-statement* completes, execution continues with the statement after the statement that caused the error.

EXIT
> Specifies that after *SQL-procedure-statement* completes, execution continues at the end of the compound statement that contains the handler.

Example: CONTINUE handler: This handler sets flag at_end when no more rows satisfy a query. The handler then causes execution to continue after the statement that returned no rows.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET at_end=1;
```

Example: EXIT handler: This handler places the string 'Table does not exist' into output parameter OUT_BUFFER when condition NO_TABLE occurs. NO_TABLE is previously declared as SQLSTATE 42704 (*name* is an undefined name). The handler then causes the SQL procedure to exit the compound statement in which the handler is declared.

```
DECLARE NO_TABLE CONDITION FOR '42704';
 :
 :
DECLARE EXIT HANDLER FOR NO_TABLE
 SET OUT_BUFFER='Table does not exist';
```

Referencing the SQLCODE and SQLSTATE values: When an SQL error or warning occurs in an SQL procedure, you might need to reference the SQLCODE or SQLSTATE values in your SQL procedure or pass those values to the procedure caller. Before you can reference SQLCODE or SQLSTATE values, you must declare the SQLCODE and SQLSTATE as SQL variables. The definitions are:

```
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
```

If you want to pass the SQLCODE or SQLSTATE values to the caller, your SQL procedure definition needs to include output parameters for those values. After an error occurs, and before control returns to the caller, you can assign the value of SQLCODE or SQLSTATE to the corresponding output parameter. For example, you might include assignment statements in an SQLEXCEPTION handler to assign the SQLCODE value to an output parameter:

```
CREATE PROCEDURE UPDATESALARY1
 (IN EMPNUMBR CHAR(6),
  OUT SQLCPARM INTEGER)
 LANGUAGE SQL
 :
 :
BEGIN:
 DECLARE SQLCODE INTEGER;
 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  SET SQLCPARM = SQLCODE;
 :
 :
```

Every statement in an SQL procedure sets the SQLCODE and SQLSTATE. Therefore, if you need to preserve SQLCODE or SQLSTATE values after a statement executes, use a simple assignment statement to assign the SQLCODE and SQLSTATE values to other variables. For example, a statement like the following does not preserve the SQLCODE value:

```
IF (1=1) THEN SET SQLCDE = SQLCODE;
```

Because the IF statement is true, the SQLCODE value is reset to zero, and you lose the previous SQLCODE value.

Handling truncation errors in an SQL procedure: Truncation during any of the following assignments in an SQL procedure causes the SQL procedure to end unless a CONTINUE handler is defined:
- Assignment of a value to an SQL variable or parameter
- Specification of a default value in a DECLARE statement

You can declare a general CONTINUE for SQLEXCEPTION, or you can declare the specific CONTINUE handlers for the following SQLSTATE values:

22001 For character truncation

22003 For numeric truncation

Examples of SQL procedures

This section contains examples of how to use each of the statements that can appear in an SQL procedure body.

Example: CASE statement: The following SQL procedure demonstates how to use a CASE statement. The procedure receives an employee's ID number and rating as input parameters. The CASE statement modifies the employee's salary and bonus, using a different UPDATE statement for each of the possible ratings.

```
CREATE PROCEDURE UPDATESALARY2
 (IN EMPNUMBR CHAR(6),
  IN RATING INT)
 LANGUAGE SQL
 MODIFIES SQL DATA
 CASE RATING
  WHEN 1 THEN
   UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY * 1.10, BONUS = 1000
    WHERE EMPNO = EMPNUMBR;
  WHEN 2 THEN
   UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY * 1.05, BONUS = 500
    WHERE EMPNO = EMPNUMBR;
  ELSE
   UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY * 1.03, BONUS = 0
    WHERE EMPNO = EMPNUMBR;
 END CASE
```

Example: Compound statement with nested IF and WHILE statements: The following example shows a compound statement that includes an IF statement, a WHILE statement, and assignment statements. The example also shows how to declare SQL variables, cursors, and handlers for classes of error codes.

The procedure receives a department number as an input parameter. A WHILE statement in the procedure body fetches the salary and bonus for each employee in the department, and uses an SQL variable to calculate a running total of employee salaries for the department. An IF statement within the WHILE statement tests for positive bonuses and increments an SQL variable that counts the number of bonuses in the department. When all employee records in the department have been processed, the FETCH statement that retrieves employee records receives SQLCODE 100. A NOT FOUND condition handler makes the search condition for the WHILE statement false, so execution of the WHILE statement ends. Assignment statements then assign the total employee salaries and the number of bonuses for the department to the output parameters for the stored procedure.

If any SQL statement in the procedure body receives a negative SQLCODE, the SQLEXCEPTION handler receives control. This handler sets output parameter DEPTSALARY to NULL and ends execution of the SQL procedure. When this handler is invoked, the SQLCODE and SQLSTATE are set to 0.

```
CREATE PROCEDURE RETURNDEPTSALARY
 (IN DEPTNUMBER CHAR(3),
  OUT DEPTSALARY DECIMAL(15,2),
  OUT DEPTBONUSCNT INT)
 LANGUAGE SQL
 READS SQL DATA
 P1: BEGIN
     DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
     DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
     DECLARE TOTAL_SALARY DECIMAL(15,2) DEFAULT 0;
     DECLARE BONUS_CNT INT DEFAULT 0;
```

```
                 DECLARE END_TABLE INT DEFAULT 0;
                 DECLARE C1 CURSOR FOR
                  SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
                   WHERE WORKDEPT = DEPTNUMBER;
                 DECLARE CONTINUE HANDLER FOR NOT FOUND
                  SET END_TABLE = 1;
                 DECLARE EXIT HANDLER FOR SQLEXCEPTION
                  SET DEPTSALARY = NULL;
                 OPEN C1;
                 FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
                 WHILE END_TABLE = 0 DO
                  SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
                  IF EMPLOYEE_BONUS > 0 THEN
                   SET BONUS_CNT = BONUS_CNT + 1;
                  END IF;
                  FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
                 END WHILE;
                 CLOSE C1;
                 SET DEPTSALARY = TOTAL_SALARY;
                 SET DEPTBONUSCNT = BONUS_CNT;
               END P1
```

Example: Compound statement with dynamic SQL statements: The following
example shows a compound statement that includes dynamic SQL statements.

The procedure receives a department number (P_DEPT) as an input parameter. In
the compound statement, three statement strings are built, prepared, and executed.
The first statement string executes a DROP statement to ensure that the table to be
created does not already exist. This table is named DEPT_*deptno*_T, where *deptno*
is the value of input parameter P_DEPT. The next statement string executes a
CREATE statement to create DEPT_*deptno*_T. The third statement string inserts
rows for employees in department *deptno* into DEPT_*deptno*_T. Just as statement
strings that are prepared in host language programs cannot contain host variables,
statement strings in SQL procedures cannot contain SQL variables or stored
procedure parameters. Therefore, the third statement string contains a parameter
marker that represents P_DEPT. When the prepared statement is executed,
parameter P_DEPT is substituted for the parameter marker.

```
CREATE PROCEDURE CREATEDEPTTABLE (IN P_DEPT CHAR(3))
 LANGUAGE SQL
 BEGIN
  DECLARE STMT CHAR(1000);
  DECLARE MESSAGE CHAR(20);
  DECLARE TABLE_NAME CHAR(30);
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
   SET MESSAGE = 'ok';
  SET TABLE_NAME = 'DEPT_'||P_DEPT||'_T';
  SET STMT = 'DROP TABLE '||TABLE_NAME;
  PREPARE S1 FROM STMT;
  EXECUTE S1;
  SET STMT = 'CREATE TABLE '||TABLE_NAME||
   '( EMPNO CHAR(6) NOT NULL, '||
   'FIRSTNME VARCHAR(6) NOT NULL, '||
   'MIDINIT CHAR(1) NOT NULL, '||
   'LASTNAME CHAR(15) NOT NULL, '||
   'SALARY DECIMAL(9,2))';
  PREPARE S2 FROM STMT;
  EXECUTE S2;
  SET STMT = 'INSERT INTO '||TABLE_NAME ||
   'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY '||
   'FROM EMPLOYEE '||
   'WHERE WORKDEPT = ?';
  PREPARE S3 FROM STMT;
  EXECUTE S3 USING P_DEPT;
 END
```

Preparing an SQL procedure

After you create the source statements for an SQL procedure, you need to prepare the procedure to run. This process involves two basic tasks:
- Creating an executable load module and a DB2 package from the SQL procedure source statements

  This task includes the following steps:
  – Precompiling the C language source program to generate a DBRM and a modified C source program
  – Binding the DBRM to generate a DB2 package
- Defining the stored procedure to DB2

  This is done by executing the CREATE PROCEDURE statement for the SQL procedure statically or dynamically. If you prepare an SQL procedure through the SQL procedure processor or the IBM DB2 Stored Procedure Builder, this task is performed for you.

There are three methods available for preparing an SQL procedure to run:
- Using IBM DB2 Stored Procedure Builder, which runs on Windows NT, Windows 95, or Windows 98.
- Using JCL. See "Using JCL to prepare an SQL procedure".
- Using the DB2 for OS/390 and z/OS SQL procedure processor. See "Using the DB2 for OS/390 and z/OS SQL procedure processor to prepare an SQL procedure" on page 562.

To run an SQL procedure, you must call it from a client program, using the SQL CALL statement. See the description of the CALL statement in Chapter 5 of *DB2 SQL Reference* for more information.

Using JCL to prepare an SQL procedure

Use the following steps to prepare an SQL procedure using JCL.

1. Preprocess the CREATE PROCEDURE statement.

   To do this, execute program DSNHPC, with the HOST(SQL) option. This process converts the SQL procedure source statements into a C language program.

2. Precompile the C language source program that was generated in step 1.

   This process produces a DBRM and modified C language source statements.

   When you perform this step, ensure that you do the following things:
   - Give the DBRM the same name as the name of the load module for the SQL procedure.
   - Specify MARGINS(1,80) for the MARGINS precompiler option.

3. Compile and link-edit the modified C source statements that were produced in step 1.

   This process produces an executable C language program.

   When you compile the C language program, ensure that the compiler options include the options MARGINS(1,80) and NOSEQ.

4. Bind the DBRM that was produced in step 1 into a package.

5. Define the stored procedure to DB2.

   To do this, execute the CREATE PROCEDURE statement for the SQL procedure. You can embed the CREATE PROCEDURE statement in an application program or execute the statement dynamically, using an application such as SPUFI or DSNTEP2. Executing the CREATE PROCEDURE statement puts the stored procedure definition in the DB2 catalog.

Using the DB2 for OS/390 and z/OS SQL procedure processor to prepare an SQL procedure

The SQL procedure processor, DSNTPSMP, is a REXX stored procedure that you can use to prepare an SQL procedure for execution. You can also use DSNTPSMP to perform selected steps in the preparation process or delete an existing SQL procedure.

The following sections contain information on invoking DSNTPSMP.

Environment for calling and running DSNTPSMP: You can invoke DSNTPSMP only through an SQL CALL statement in an application program or through IBM DB2 Stored Procedure Builder.

Before you can run DSNTPSMP, you need to perform the following steps to set up the DSNTPSMP environment:

1. Install DB2 for OS/390 and z/OS REXX Language Support feature.

   Contact your IBM service representative for more information.

2. If you plan to call DSNTPSMP directly, write and prepare an application program that executes an SQL CALL statement for DSNTPSMP.

   See "Writing and preparing an application that calls DSNTPSMP" on page 564 for more information.

   If you plan to invoke DSNTPSMP through the IBM DB2 Stored Procedure Builder, see the following URL for information on installing and using the IBM DB2 Stored Procedure Builder.

   `http://www.ibm.com/software/data/db2/os390/spb`

3. Set up a WLM environment in which to run DSNTPSMP. See Part 5 (Volume 2) of *DB2 Administration Guide* for general information on setting up WLM application environments for stored procedures and "Setting up a WLM application environment for DSNTPSMP" for specific information for DSNTPSMP.

Setting up a WLM application environment for DSNTPSMP: You must run DSNTPSMP in a WLM-established stored procedures address space. You should run only DSNTPSMP in that address space, and you should not run multiple copies of DSNTPSMP concurrently.

Figure 170 on page 563 shows sample JCL for a startup procedure for the address space in which DSNTPSMP runs.

```
//DSNWLM   PROC RGN=0K,APPLENV=WLMTEST,DB2SSN=DSN,NUMTCB=1        1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//         PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD                    2
//         DD  DISP=SHR,DSN=CBC.SCBCCMP
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DSN710.SDSNLOAD
//SYSEXEC  DD  DISP=SHR,DSN=DSN710.SDSNCLST                       3
//SYSTSPRT DD  SYSOUT=A
//CEEDUMP  DD  SYSOUT=A
//SYSPRINT DD  SYSOUT=A
//SYSABEND DD  DUMMY
//SQLDBRM  DD  DISP=SHR,DSN=DSN710.DBRMLIB.DATA                   4
//SQLCSRC  DD  DISP=SHR,DSN=USER.PSMLIB.DATA                      5
//SQLLMOD  DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD                    6
//SQLLIBC  DD  DISP=SHR,DSN=CEE.SCEEH.H                           7
//SQLLIBL  DD  DISP=SHR,DSN=CEE.SCEELKED                          8
//         DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=DSN710.SDSNEXIT
//         DD  DISP=SHR,DSN=DSN710.SDSNLOAD
//SYSMSGS  DD  DISP=SHR,DSN=CEE.SCEEMSGP(EDCPMSGE)                9
//SQLSRC   DD  UNIT=SYSDA,SPACE=(400,(20,20)),                   10
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLPRINT DD  SPACE=(16000,(20,20)),UNIT=SYSDA,
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLTERM  DD  SPACE=(16000,(20,20)),UNIT=SYSDA,
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLOUT   DD  SPACE=(16000,(20,20)),UNIT=SYSDA,
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLCPRT  DD  SPACE=(16000,(20,20)),UNIT=SYSDA,
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLUT1   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLUT2   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLCIN   DD  UNIT=SYSDA,SPACE=(800,(&WSPC,&WSPC))
//SQLLIN   DD  UNIT=SYSDA,SPACE=(8000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//SQLWORK1 DD  SPACE=(800,(&WSPC,&WSPC)),UNIT=SYSDA,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=4560)
//SQLWORK2 DD  SPACE=(400,(20,20)),UNIT=SYSDA,
//             DCB=(RECFM=U,BLKSIZE=32760)
//SYSMOD   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

Figure 170. Startup procedure for a WLM address space in which DSNTPSMP runs

Notes to Figure 170:

1 APPLENV specifies the application environment in which DSNTPSMP runs. To ensure that DSNTPSMP always uses the correct data sets and parameters for preparing each SQL procedure, you can set up different application environments for preparing different types of SQL procedures. For example, if all payroll applications use the same set of data sets during program preparation, you could set up an application environment called PAYROLL for preparing only payroll applications. The startup procedure for PAYROLL would point to the data sets that are used for payroll applications.

DB2SSN specifies the DB2 subsystem name.

NUMTCB specifies the number of programs that can run concurrently in the address space. You should always set NUMTCB to 1 to ensure that executions of DSNTPSMP occur serially.

2	STEPLIB specifies the Language Environment run-time library that DSNTPSMP uses when it runs.
3	SYSEXEC specifies the library that contains DSNTPSMP.
4	DBRMLIB specifies the library into which DSNTPSMP puts the DBRM that it generates when it precompiles your SQL procedure.
5	SQLCSRC specifies the library into which DSNTPSMP puts the C source code that it generates from the SQL procedure source code. This data set should have a logical record length of 80.
6	SQLLMOD specifies the library into which DSNTPSMP puts the load module that it generates when it compiles and link-edits your SQL procedure.
7	SQLLIBC specifies the library that contains standard C header files. This library is used during compilation of the generated C program.
8	SQLLIBL specifies the following libraries, which DSNTPSMP uses when it link-edits the SQL procedure: • Language Environment run-time library • DB2 application load library • DB2 exit library • DB2 load library
9	SYSMSGS specifies the library that contains messages that are used by the C prelink-edit utility.
10	The DD statements that follow describe work file data sets that are used by DSNTPSMP.

Authorization to execute DSNTPSMP: The program that invokes DSNTPSMP must have the following authorizations:

- Authorization to execute the CALL statement. See the description of the CALL statement in Chapter 5 of *DB2 SQL Reference* for more information.
- The BIND privilege for any stored procedure packages that DSNTPSMP binds.

Writing and preparing an application that calls DSNTPSMP: DSNTPSMP must be invoked through an SQL CALL statement in an application program. This section contains information that you need to write and prepare the calling application.

DSNTPSMP Syntax:
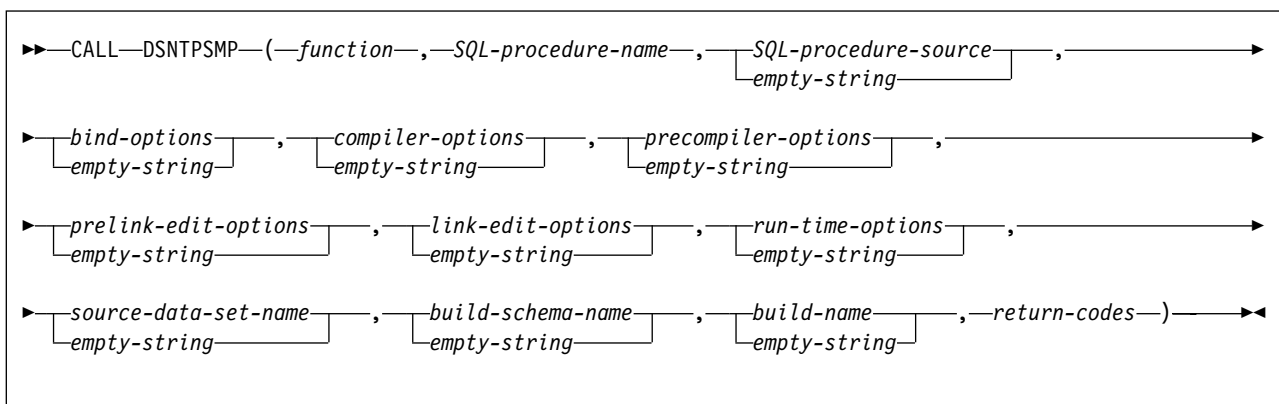
bind-options, compiler-options, precompiler-options, prelink-edit-options, link-edit options, or

run-time-options:

```
>>--'--+-------+--'----------------------------------------><
        |  ,<-- |
        +-option-+
```

DSNTPSMP parameters:

function
> A VARCHAR(20) input parameter that identifies the task that you want
> DSNTPSMP to perform. The tasks are:

> **BUILD**
>> Creates the following objects for an SQL procedure:
>> - A DBRM, in the data set that DD name SQLDBRM points to
>> - A load module, in the data set that DD name SQLLMOD points to
>> - The C language source code for the SQL procedure, in the data set that
>>   DD name SQLCSRC points to
>> - The stored procedure package
>> - The stored procedure definition
>>
>> If you choose the create function, and an SQL procedure with name
>> *SQL-procedure-name* already exists, DSNTPSMP issues a warning
>> message and terminates.

> **DESTROY**
>> Deletes the following objects for an SQL procedure:
>> - The DBRM, from the data set that DD name SQLDBRM points to
>> - The load module, from the data set that DD name SQLLMOD points to
>> - The C language source code for the SQL procedure, from the data set
>>   that DD name SQLCSRC points to
>> - The stored procedure package
>> - The stored procedure definition
>>
>> Before the DESTROY function can execute successfully, you must execute
>> DROP PROCEDURE on the SQL procedure.

> **REBUILD**
>> Replaces all objects that were created by the BUILD function.

> **REBIND**
>> Rebinds an SQL procedure package.

SQL-procedure-name
> A VARCHAR(18) input parameter performs the following functions:
> - Specifies the SQL procedure name for the DESTROY or REBIND function
> - Specifies the name of the SQL procedure load module for the BUILD or
>   REBUILD function

SQL-procedure-source
> A VARCHAR(32672) input parameter that contains the source code for the SQL

procedure. If you specify an empty string for this parameter, you need to specify the name of a data set that contains the SQL procedure source code, in *source-data-set-name*.

bind-options
A VARCHAR(1024) input parameter that contains the options that you want to specify for binding the SQL procedure package. For a list of valid bind options, see Chapter 2 of *DB2 Command Reference*.

You must specify the PACKAGE bind option for the BUILD, REBUILD, and REBIND functions.

compiler-options
A VARCHAR(255) input parameter that contains the options that you want to specify for compiling the C language program that DB2 generates for the SQL procedure. For a list of valid compiler options, see *OS/390 C/C++ User's Guide*.

precompiler-options
A VARCHAR(255) input parameter that contains the options that you want to specify for precompiling the C language program that DB2 generates for the SQL procedure. For a list of valid precompiler options, see Part 5 of *DB2 Application Programming and SQL Guide*.

prelink-edit-options
A VARCHAR(255) input parameter that contains the options that you want to specify for prelink-editing the C language program that DB2 generates for the SQL procedure. For a list of valid prelink-edit options, see *OS/390 C/C++ User's Guide*.

link-edit-options
A VARCHAR(255) input parameter that contains the options that you want to specify for link-editing the C language program that DB2 generates for the SQL procedure. For a list of valid link-edit options, see *DFSMS/MVS: Program Management*.

run-time-options
A VARCHAR(254) input parameter that contains the Language Environment run-time options that you want to specify for the SQL procedure. For a list of valid Language Environment run-time options, see *OS/390 Language Environment for OS/390 & VM Programming Reference*.

source-data-set-name
A VARCHAR(80) input parameter that contains the name of an MVS sequential data set or partitioned data set member that contains the source code for the SQL procedure. If you specify an empty string for this parameter, you need to provide the SQL procedure source code in *SQL-procedure-source*.

build-schema-name
A VARCHAR(8) input parameter that contains the schema name for the procedure name that you specify for the *build-name* parameter.

build-name
A VARCHAR(18) input parameter that contains the procedure name that you use when you call DSNTPSMP. You might create several stored procedure definitions for DSNTPSMP, each of which specifies a different WLM environment. When you call DSNTPSMP using the name in this parameter, DB2 runs DSNTPSMP in the WLM environment that is associated with the procedure name.

return-codes
>A VARCHAR(255) output parameter in which DB2 puts the return codes from all steps of the DSNTPSMP invocation.

Result sets that DSNTPSMP returns: When errors occur during DSNTPSMP execution, DB2 returns a result set that contains messages and listings from each step that DSNTPSMP performs. To obtain the information from the result set, you can write your client program to retrieve information from one result set with known contents. However, for greater flexibility, you might want to write your client program to retrieve data from an unknown number of result sets with unknown contents. Both techniques are shown in "Writing a DB2 for OS/390 and z/OS client program or SQL procedure to receive result sets" on page 599.

Each row of the result set contains the following information:

Processing step
>The step in the *function* process to which the message applies.

ddname
>The ddname of the data set that contains the message.

Sequence number
>The sequence number of a line of message text within a message.

Message
>A line of message text.

Rows in the message result set are ordered by processing step, ddname, and sequence number.

Examples of DSNTPSMP invocation: **DSNTPSMP BUILD function:** Call DSNTPSMP to build an SQL procedure. The information that DSNTPSMP needs is:

Function	BUILD
Source location	String in variable procsrc
Bind options	SQLERROR(NOPACKAGE), VALIDATE(RUN), ISOLATION(RR), RELEASE(COMMIT)
Compiler options	SOURCE, LIST, MAR(1,80), LONGNAME, RENT
Precompiler options	HOST(SQL), SOURCE, XREF, MAR(1,72), STDSQL(NO)
Prelink-edit options	None specified
Link-edit options	AMODE=31, RMODE=ANY, MAP, RENT
Run-time options	MSGFILE(OUTFILE), RPTSTG(ON), RPTOPTS(ON)
Build schema	MYSCHEMA
Build name	WLM2PSMP

The CALL statement is:
```
EXEC SQL CALL DSNTPSMP('BUILD','',procsrc,
 'SQLERROR(NOPACKAGE),VALIDATE(RUN),ISOLATION(RR),RELEASE(COMMIT)',
 'SOURCE,LIST,MAR(1,80),LONGNAME,RENT',
 'HOST(SQL),SOURCE,XREF,MAR(1,72),STDSQL(NO)',
 '',
 'AMODE=31,RMODE=ANY,MAP,RENT',
 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)',
 'MYSCHEMA',
 'WLM2PSMP',
 '');
```

DSNTPSMP DESTROY function: Call DSNTPSMP to delete an SQL procedure definition and the associated load module. The information that DSNTPMSP needs

is:

Function   DESTROY
SQL procedure name   OLDPROC

The CALL statement is:
```
EXEC SQL CALL DSNTPSMP('DESTROY','OLDPROC','',
 '','','','','','',
 '','','');
```

DSNTPSMP REBUILD function: Call DSNTPSMP to recreate an existing SQL
procedure. The information that DSNTPMSP needs is:

Function   REBUILD
Source location   Member PROCSRC of partitioned data set DSN710.SDSNSAMP
Bind options   SQLERROR(NOPACKAGE), VALIDATE(RUN), ISOLATION(RR),
                        RELEASE(COMMIT)
Compiler options   SOURCE, LIST, MAR(1,80), LONGNAME, RENT
Precompiler options   HOST(SQL), SOURCE, XREF, MAR(1,72), STDSQL(NO)
Prelink-edit options   MAP
Link-edit options   AMODE=31, RMODE=ANY, MAP, RENT
Run-time options   MSGFILE(OUTFILE), RPTSTG(ON), RPTOPTS(ON)

The CALL statement is:
```
EXEC SQL CALL DSNTPSMP('REBUILD','','',
 'SQLERROR(NOPACKAGE),VALIDATE(RUN),ISOLATION(RR),RELEASE(COMMIT)',
 'SOURCE,LIST,MAR(1,80),LONGNAME,RENT',
 'HOST(SQL),SOURCE,XREF,MAR(1,72),STDSQL(NO)',
 'MAP',
 'AMODE=31,RMODE=ANY,MAP,RENT',
 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)','','',
 'DSN710.SDSNSAMP(PROCSRC)');
```

DSNTPSMP REBIND function: Call DSNTPSMP to rebind the package for an
existing SQL procedure. The information that DSNTPMSP needs is:

Function   REBIND
SQL procedure name   SQLPROC
Bind options   VALIDATE(BIND), ISOLATION(RR), RELEASE(DEALLOCATE)

The CALL statement is:
```
EXEC SQL CALL DSNTPSMP('REBIND','SQLPROC','',
 'VALIDATE(BIND),ISOLATION(RR),RELEASE(DEALLOCATE)',
 '','','','','',
 '','','');
```

Preparing a program that invokes DSNTPSMP: To prepare the program that calls
DSNTPSMP for execution, you need to perform the following steps:

1. Precompile, compile, and link-edit the application program.
2. Bind a package for the application program.
3. Bind the package for DB2 REXX support, DSNTRXCS.DSNTREXX, and the
   package for the application program into a plan.

Sample programs to help you prepare and run SQL procedures

Table 61 lists the sample jobs that DB2 provides to help you prepare and run SQL procedures. All samples are in data set DSN710.SDSNSAMP. Before you can run the samples, you must customize them for your installation. See the prolog of each sample for specific instructions.

Table 61. SQL procedure samples shipped with DB2

Member that contains source code	Contents	Purpose
DSNHSQL	JCL procedure	Precompiles, compiles, prelink-edits, and link-edits an SQL procedure
DSNTEJ63	JCL job	Invokes JCL procedure DSNHSQL to prepare SQL procedure DSN8ES1 for execution
DSN8ES1	SQL procedure	A stored procedure that accepts a department number as input and returns a result set that contains salary information for each employee in that department
DSNTEJ64	JCL job	Prepares client program DSN8ED3 for execution
DSN8ED3	C program	Calls SQL procedure DSN8ES1
DSN8ES2	SQL procedure	A stored procedure that accepts one input parameter and returns two output parameters. The input parameter specifies a bonus to be awarded to managers. The SQL procedure updates the BONUS column of DSN710.SDSNSAMP. If no SQL error occurs when the SQL procedure runs, the first output parameter contains the total of all bonuses awarded to managers and the second output parameter contains a null value. If an SQL error occurs, the second output parameter contains an SQLCODE.
DSN8ED4	C program	Calls the SQL procedure processor, DSNTPSMP, to prepare DSN8ES2 for execution
DSN8WLM	JCL procedure	A sample startup procedure for the WLM-established stored procedures address space in which DSNTPSMP runs
DSN8ED5	C program	Calls SQL procedure DSN8ES2
DSNTEJ65	JCL job	Prepares and executes programs DSN8ED4 and DSN8ED5

Writing and preparing an application to use stored procedures

Use the SQL statement CALL to call a stored procedure and to pass a list of parameters to the procedure. See Chapter 5 of *DB2 SQL Reference* for the syntax and a complete description of the CALL statement.

An application program that calls a stored procedure can:

- Call more than one stored procedure
- Execute the CALL statement locally or send the CALL statement to a server. The application executes a CONNECT statement to connect to the server and then executes the CALL statement, or uses a 3-part name to identify and implicitly connect to the server where the stored procedure is located.
- After connecting to a server, mix CALL statements with other SQL statements.

Use either of these methods to execute the CALL statement:

- Execute the CALL statement statically.
  - Use an escape clause in an ODBC application to pass the CALL statement to DB2.
- Use any of the DB2 attachment facilities

Forms of the CALL statement

The simplest form of a CALL statement looks like this:

```
EXEC SQL CALL A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CODE);
```

where :EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, and :CODE are host variables that you have declared earlier in your application program. Your CALL statement might vary from the above statement in the following ways:

- Instead of passing each of the employee and project parameters separately, you could pass them together as a host structure. For example, if you define a host structure like this:

```
struct {
  char EMP[7];
  char PRJ[7];
  short ACT;
  short EMT;
  char  EMS[11];
  char  EME[11];
} empstruc;
```

the CALL statement looks like this:

```
EXEC SQL CALL A (:empstruc, :TYPE, :CODE);
```

- Suppose that A is in schema SCHEMAA at remote location LOCA. To access A, you could use either of these methods:
  - Execute a CONNECT statement to LOCA and then execute the CALL statement:

```
CONNECT TO LOCA;
EXEC SQL CALL SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,
  :TYPE, :CODE);
```

  - Specify the 3-part name for A in the CALL statement:

```
EXEC SQL CALL LOCA.SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS,
  :EME, :TYPE, :CODE);
```

The advantage of using the second form is that you do not need to execute a CONNECT statement. The disadvantage is that this form of the CALL statement is not portable to other platforms.

If your program executes the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statements, you must use the same form of the procedure name on the CALL statement and on the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statement.

- The examples above assume that none of the input parameters can have null values. To allow null values, code a statement like this:

```
EXEC SQL CALL A (:EMP :IEMP, :PRJ :IPRJ, :ACT :IACT,
                 :EMT :IEMT, :EMS :IEMS, :EME :IEME,
                 :TYPE :ITYPE, :CODE :ICODE);
```

where :IEMP, :IPRJ, :IACT, :IEMT, :IEMS, :IEME, :ITYPE, and :ICODE are indicator variables for the parameters.

- You might pass integer or character string constants or the null value to the stored procedure, as in this example:

```
EXEC SQL CALL A ('000130', 'IF1000', 90, 1.0, NULL, '1982-10-01',
                 :TYPE, :CODE);
```

- You might use a host variable for the name of the stored procedure:

```
EXEC SQL CALL :procnm (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,
  :TYPE, :CODE);
```

Assume that the stored procedure name is 'A'. The host variable *procnm* is a character variable of length 255 or less that contains the value 'A'. You should use this technique if you do not know in advance the name of the stored procedure, but you do know the parameter list convention.

- If you prefer to pass your parameters in a single structure, rather than as separate host variables, you might use this form:

```
EXEC SQL CALL A USING DESCRIPTOR :sqlda;
```

sqlda is the name of an SQLDA.

One advantage of using this form is that you can change the encoding scheme of the stored procedure parameter values. For example, if the subsystem on which the stored procedure runs has an EBCDIC encoding scheme, and you want to retrieve data in ASCII CCSID 437, you can specify the desired CCSIDs for the output parameters in the SQLVAR fields of the SQLDA. The technique for overriding the CCSIDs of parameters is the same as the technique for overriding the CCSIDs of variables, which is described in "Changing the CCSID for retrieved data" on page 517. When you use this technique, the defined encoding scheme of the parameter must be different from the encoding scheme that you specify in the SQLDA. Otherwise, no conversion occurs. The defined encoding scheme for the parameter is the encoding scheme that you specify in the CREATE PROCEDURE statement, or the default encoding scheme for the subsystem, if you do not specify an encoding scheme in the CREATE PROCEDURE statement.

- You might execute the CALL statement by using a host variable name for the stored procedure with an SQLDA:

```
EXEC SQL CALL :procnm USING DESCRIPTOR :sqlda;
```

This form gives you extra flexibility because you can use the same CALL statement to call different stored procedures with different parameter lists.

Your client program must assign a stored procedure name to the host variable *procnm* and load the SQLDA with the parameter information before making the SQL CALL.

Each of the above CALL statement examples uses an SQLDA. If you do not explicitly provide an SQLDA, the precompiler generates the SQLDA based on the variables in the parameter list.

Authorization for executing stored procedures

To execute a stored procedure, you need two types of authorization:

- Authorization to execute the CALL statement
- Authorization to execute the stored procedure package and any packages under the stored procedure package.

The authorizations you need depend on whether the form of the CALL statement is CALL *literal* or CALL :*host-variable.*

If the stored procedure invokes user-defined functions or triggers, you need additional authorizations to execute the trigger, the user-defined function, and the user-defined function packages.

For more information, see the description of the CALL statement in Chapter 5 of *DB2 SQL Reference.*

Linkage conventions

When an application executes the CALL statement, DB2 builds a parameter list for the stored procedure, using the parameters and values provided in the statement. DB2 obtains information about parameters from the stored procedure definition you create when you execute CREATE PROCEDURE. Parameters are defined as one of these types:

IN   Input-only parameters, which provide values to the stored procedure

OUT   Output-only parameters, which return values from the stored procedure to the calling program

INOUT
Input/output parameters, which provide values to or return values from the stored procedure.

If a stored procedure fails to set one or more of the output-only parameters, DB2 does not detect the error in the stored procedure. Instead, DB2 returns the output parameters to the calling program, with the values established on entry to the stored procedure.

Initializing output parameters: For a stored procedure that runs locally, you do not need to initialize the values of output parameters before you call the stored procedure. However, when you call a stored procedure at a remote location, the local DB2 cannot determine whether the parameters are input (IN) or output (OUT or INOUT) parameters. Therefore, you must initialize the values of all output parameters before you call a stored procedure at a remote location.

It is recommended that you initialize the length of LOB output parameters to zero. Doing so can improve your performance.

DB2 supports three parameter list conventions. DB2 chooses the parameter list convention based on the value of the PARAMETER STYLE parameter in the stored procedure definition: **GENERAL**, **GENERAL WITH NULLS**, or **DB2SQL**.

- **GENERAL:** Use GENERAL when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure. The stored procedure must contain a variable declaration for each parameter passed in the CALL statement.

    Figure 171 on page 573 shows the structure of the parameter list for PARAMETER STYLE GENERAL.
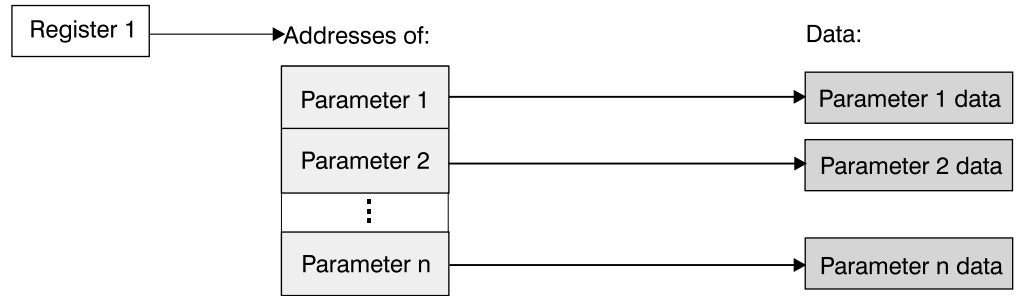
Figure 171. Parameter convention GENERAL for a stored procedure

- **GENERAL WITH NULLS:** Use GENERAL WITH NULLS to allow the calling program to supply a null value for any parameter passed to the stored procedure. For the GENERAL WITH NULLS linkage convention, the stored procedure must do the following:
  - Declare a variable for each parameter passed in the CALL statement.
  - Declare a null indicator structure containing an indicator variable for each parameter.
  - On entry, examine all indicator variables associated with input parameters to determine which parameters contain null values.
  - On exit, assign values to all indicator variables associated with output variables. An indicator variable for an output variable that returns a null value to the caller must be assigned a negative number. Otherwise, the indicator variable must be assigned the value 0.

In the CALL statement, follow each parameter with its indicator variable, using one of the forms below:

```
host-variable :indicator-variable
or
host-variable INDICATOR :indicator-variable.
```

Figure 172 shows the structure of the parameter list for PARAMETER STYLE GENERAL WITH NULLS.
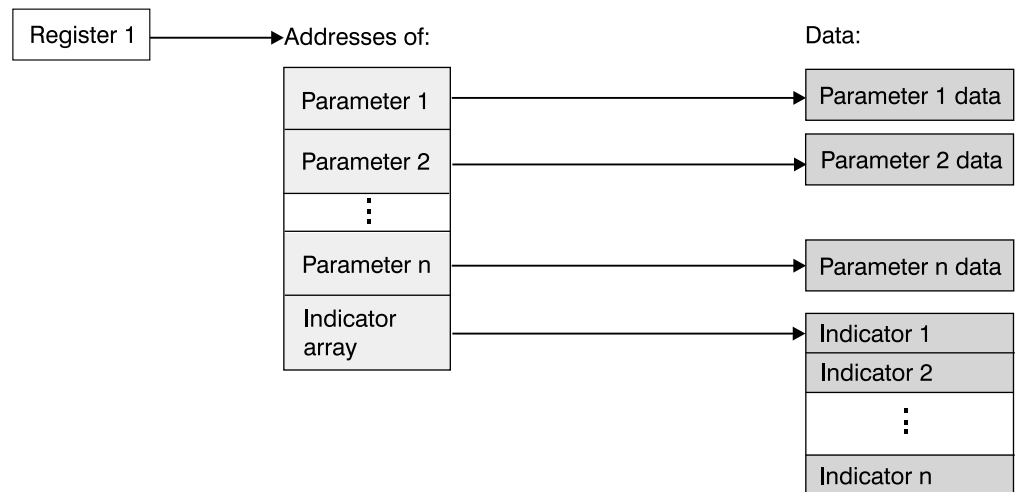
Figure 172. Parameter convention GENERAL WITH NULLS for a stored procedure

- **DB2SQL:** Like GENERAL WITH NULLS, option DB2SQL lets you supply a null value for any parameter that is passed to the stored procedure. In addition, DB2 passes input and output parameters to the stored procedure that contain this information:
  - The SQLSTATE that is to be returned to DB2. This is a CHAR(5) parameter that can have the same values as those that are returned from a user-defined function. See "Passing parameter values to and from a user-defined function" on page 251 for valid SQLSTATE values.
  - The qualified name of the stored procedure. This is a VARCHAR(27) value.
  - The specific name of the stored procedure. The specific name is a VARCHAR(18) value that is the same as the unqualified name.
  - The SQL diagnostic string that is to be returned to DB2. This is a VARCHAR(70) value. Use this area to pass descriptive information about an error or warning to the caller.

  DB2SQL is not a valid linkage convention for a REXX language stored procedure.

  Figure 173 shows the structure of the parameter list for PARAMETER STYLE DB2SQL.
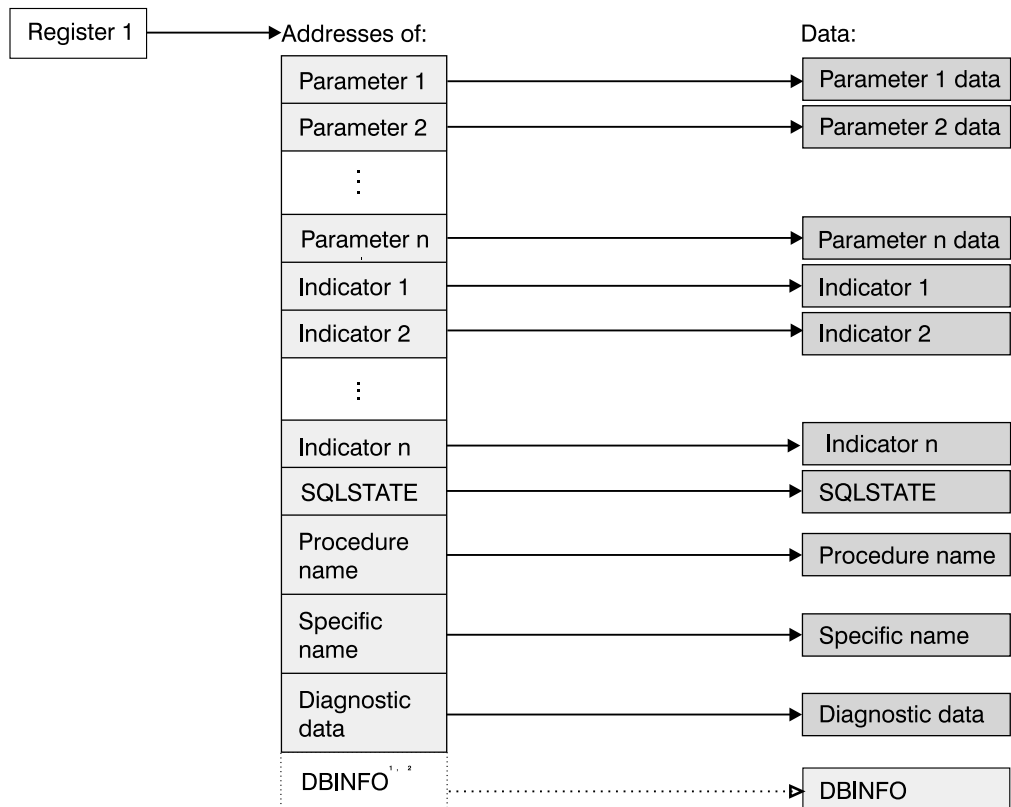
[1] For PL/I, this value is the address of a pointer to the DBINFO data.
[2] Passed if the DBINFO option is specified in the user-defined function definition

Figure 173. Parameter convention DB2SQL for a stored procedure

Example of stored procedure linkage convention GENERAL

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL linkage convention to receive parameters. See "Examples of using stored procedures" on page 890 for examples of complete stored procedures and application programs that call them.

For these examples, assume that a COBOL application has the following parameter declarations and CALL statement:

```
************************************************************
* PARAMETERS FOR THE SQL STATEMENT CALL                   *
************************************************************
01 V1  PIC S9(9) USAGE COMP.
01 V2  PIC X(9).
   :
   :
      EXEC SQL CALL A (:V1, :V2) END-EXEC.
```

In the CREATE PROCEDURE statement, the parameters are defined like this:

```
V1 INT IN, V2 CHAR(9) OUT
```

Figure 174, Figure 175, Figure 176, and Figure 177 show how a stored procedure in each language receives these parameters.

```
      ********************************************************************
      *   CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES    *
      *   THE GENERAL LINKAGE CONVENTION.                              *
      ********************************************************************
      A       CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
              USING PROGAREA,R13
      ********************************************************************
      *  BRING UP THE LANGUAGE ENVIRONMENT.                            *
      ********************************************************************

         ⋮

      ********************************************************************
      *  GET THE PASSED PARAMETER VALUES. THE GENERAL LINKAGE CONVENTION*
      *  FOLLOWS THE STANDARD ASSEMBLER LINKAGE CONVENTION:            *
      *  ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS TO THE      *
      *  PARAMETERS.                                                   *
      ********************************************************************
              L     R7,0(R1)         GET POINTER TO V1
              MVC   LOCV1(4),0(R7)   MOVE VALUE INTO LOCAL COPY OF V1

         ⋮

              L     R7,4(R1)         GET POINTER TO V2
              MVC   0(9,R7),LOCV2    MOVE A VALUE INTO OUTPUT VAR V2

         ⋮

              CEETERM  RC=0
      ********************************************************************
      *  VARIABLE DECLARATIONS AND EQUATES                             *
      ********************************************************************
      R1      EQU   1                REGISTER 1
      R7      EQU   7                REGISTER 7
      PPA     CEEPPA  ,              CONSTANTS DESCRIBING THE CODE BLOCK
              LTORG ,                PLACE LITERAL POOL HERE
      PROGAREA DSECT
              ORG   *+CEEDSASZ       LEAVE SPACE FOR DSA FIXED PART
      LOCV1   DS    F                LOCAL COPY OF PARAMETER V1
      LOCV2   DS    CL9              LOCAL COPY OF PARAMETER V2

         ⋮

      PROGSIZE EQU  *-PROGAREA
              CEEDSA  ,              MAPPING OF THE DYNAMIC SAVE AREA
              CEECAA  ,              MAPPING OF THE COMMON ANCHOR AREA
              END   A
```

Figure 174. An example of GENERAL linkage in assembler

```
#pragma runopts(PLIST(OS))
#pragma options(RENT)
#include <stdlib.h>
#include <stdio.h>
/****************************************************************/
/* Code for a C language stored procedure that uses the        */
/* GENERAL linkage convention.                                 */
/****************************************************************/
main(argc,argv)
  int argc;                        /* Number of parameters passed */
  char *argv[];                    /* Array of strings containing */
                                   /*  the parameter values      */
{
  long int locv1;                  /* Local copy of V1           */
  char locv2[10];                  /* Local copy of V2           */
                                   /*  (null-terminated)         */

   .
   .
  /****************************************************************/
  /* Get the passed parameters. The GENERAL linkage convention   */
  /* follows the standard C language parameter passing           */
  /* conventions:                                                */
  /* - argc contains the number of parameters passed             */
  /* - argv[0]   is a pointer to the stored procedure name       */
  /* - argv[1] to argv[n] are pointers to the n parameters       */
  /*   in the SQL statement CALL.                                */
  /****************************************************************/
  if(argc==3)                      /* Should get 3 parameters:   */
  {                                /* procname, V1, V2           */
    locv1 = *(int *) argv[1];
                                   /* Get local copy of V1       */

   .
   .
    strcpy(argv[2],locv2);
                                   /* Assign a value to V2       */

   .
   .
  }
}
```

Figure 175. An example of GENERAL linkage in C

```
        CBL RENT
        IDENTIFICATION DIVISION.
       *************************************************************
       * CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
       * GENERAL LINKAGE CONVENTION.                              *
       *************************************************************
        PROGRAM-ID.    A.

           :
           :
        DATA DIVISION.

           :
           :
        LINKAGE SECTION.
       *************************************************************
       * DECLARE THE PARAMETERS PASSED BY THE SQL STATEMENT      *
       *  CALL HERE.                                             *
       *************************************************************
        01 V1  PIC S9(9) USAGE COMP.
        01 V2  PIC X(9).

           :
           :
        PROCEDURE DIVISION USING V1, V2.
       *************************************************************
       * THE USING PHRASE INDICATES THAT VARIABLES  V1 AND V2    *
       *  WERE PASSED BY THE CALLING PROGRAM.                    *
       *************************************************************

           :
           :
       **************************************
       * ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
       **************************************
          MOVE '123456789' TO V2.
```

Figure 176. An example of GENERAL linkage in COBOL

```
*PROCESS SYSTEM(MVS);
A: PROC(V1, V2) OPTIONS(MAIN NOEXECOPS REENTRANT);
 /***************************************************************/
 /* Code for a PL/I language stored procedure that uses the    */
 /* GENERAL linkage convention.                                */
 /***************************************************************/
 /***************************************************************/
 /* Indicate on the PROCEDURE statement that two parameters    */
 /*  were passed by the SQL statement CALL. Then declare the   */
 /*  parameters below.                                         */
 /***************************************************************/
   DCL V1 BIN FIXED(31),
       V2 CHAR(9);

   :
   :
   V2 = '123456789';    /* Assign a value to output variable V2 */
```

Figure 177. An example of GENERAL linkage in PL/I

Example of stored procedure linkage convention GENERAL WITH NULLS

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL WITH NULLS linkage convention to receive parameters. See "Examples of using stored procedures" on page 890 for examples of complete stored procedures and application programs that call them.

For these examples, assume that a C application has the following parameter declarations and CALL statement:

```
/***********************************************************/
/* Parameters for the SQL statement CALL                   */
/***********************************************************/
  long int v1;
  char v2[10];                     /* Allow an extra byte for    */
                                   /*  the null terminator       */
/***********************************************************/
/* Indicator structure                                     */
/***********************************************************/
  struct indicators {
    short int ind1;
    short int ind2;
  } indstruc;

   .
   .
   .
  indstruc.ind1 = 0;        /* Remember to initialize the  */
                            /*  input parameter's indicator*/
                            /*  variable before executing  */
                            /*  the CALL statement          */
  EXEC SQL CALL B (:v1 :indstruc.ind1, :v2 :indstruc.ind2);

   .
   .
   .
```

In the CREATE PROCEDURE statement, the parameters are defined like this:

```
V1 INT IN, V2 CHAR(9) OUT
```

Figure 178, Figure 179, Figure 180, and Figure 181 show how a stored procedure in each language receives these parameters.

```
      ********************************************************************
      *  CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES      *
      *  THE GENERAL WITH NULLS LINKAGE CONVENTION.                     *
      ********************************************************************
      B       CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
              USING PROGAREA,R13
      ********************************************************************
      *  BRING UP THE LANGUAGE ENVIRONMENT.                             *
      ********************************************************************

         ⋮

      ********************************************************************
      *  GET THE PASSED PARAMETER VALUES. THE GENERAL WITH NULLS LINKAGE*
      *  CONVENTION IS AS FOLLOWS:                                      *
      *  ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N        *
      *  PARAMETERS ARE PASSED, THERE ARE N+1 POINTERS. THE FIRST       *
      *  N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS      *
      *  WITH THE GENERAL LINKAGE CONVENTION. THE N+1ST POINTER IS      *
      *  THE ADDRESS OF A LIST CONTAINING THE N INDICATOR VARIABLE      *
      *  VALUES.                                                        *
      ********************************************************************
              L    R7,0(R1)          GET POINTER TO V1
              MVC  LOCV1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF V1
              L    R7,8(R1)          GET POINTER TO INDICATOR ARRAY
              MVC  LOCIND(2*2),0(R7)  MOVE VALUES INTO LOCAL STORAGE
              LH   R7,LOCIND          GET INDICATOR VARIABLE FOR V1
              LTR  R7,R7              CHECK IF IT IS NEGATIVE
              BM   NULLIN             IF SO, V1 IS NULL

         ⋮

              L    R7,4(R1)          GET POINTER TO V2
              MVC  0(9,R7),LOCV2      MOVE A VALUE INTO OUTPUT VAR V2
              L    R7,8(R1)          GET POINTER TO INDICATOR ARRAY
              MVC  2(2,R7),=H(0)      MOVE ZERO TO V2'S INDICATOR VAR

         ⋮

              CEETERM  RC=0
      ********************************************************************
      *  VARIABLE DECLARATIONS AND EQUATES                              *
      ********************************************************************
      R1      EQU  1                 REGISTER 1
      R7      EQU  7                 REGISTER 7
      PPA     CEEPPA ,               CONSTANTS DESCRIBING THE CODE BLOCK
              LTORG ,                PLACE LITERAL POOL HERE
      PROGAREA DSECT
              ORG  *+CEEDSASZ         LEAVE SPACE FOR DSA FIXED PART
      LOCV1   DS   F                 LOCAL COPY OF PARAMETER V1
      LOCV2   DS   CL9               LOCAL COPY OF PARAMETER V2
      LOCIND  DS   2H                LOCAL COPY OF INDICATOR ARRAY

         ⋮

      PROGSIZE EQU  *-PROGAREA
              CEEDSA ,               MAPPING OF THE DYNAMIC SAVE AREA
              CEECAA ,               MAPPING OF THE COMMON ANCHOR AREA
              END  B
```

Figure 178. An example of GENERAL WITH NULLS linkage in assembler

```
#pragma options(RENT)
#pragma runopts(PLIST(OS))
#include <stdlib.h>
#include <stdio.h>
/****************************************************************/
/* Code for a C language stored procedure that uses the        */
/* GENERAL WITH NULLS linkage convention.                      */
/****************************************************************/
main(argc,argv)
  int argc;                      /* Number of parameters passed */
  char *argv[];                  /* Array of strings containing */
                                 /*  the parameter values       */
{
  long int locv1;                /* Local copy of V1            */
  char locv2[10];                /* Local copy of V2            */
                                 /*  (null-terminated)          */
  short int locind[2];           /* Local copy of indicator     */
                                 /*  variable array             */
  short int *tempint;            /* Used for receiving the      */
                                 /*  indicator variable array   */

 .
 .
 .

  /****************************************************************/
  /* Get the passed parameters. The GENERAL WITH NULLS linkage   */
  /* convention is as follows:                                   */
  /* - argc contains the number of parameters passed             */
  /* - argv[0] is a pointer to the stored procedure name         */
  /* - argv[1] to argv[n] are pointers to the n parameters    */
  /*    in the SQL statement CALL.                               */
  /* - argv[n+1] is a pointer to the indicator variable array    */
  /****************************************************************/
  if(argc==4)                    /* Should get 4 parameters:    */
  {                              /*  procname, V1, V2,          */
                                 /*  indicator variable array   */
    locv1 = *(int *) argv[1];
                                 /* Get local copy of V1        */
    tempint = argv[3];           /* Get pointer to indicator    */
                                 /*  variable array             */
    locind[0] = *tempint;
                                 /* Get 1st indicator variable  */
    locind[1] = *(++tempint);
                                 /* Get 2nd indicator variable  */
    if(locind[0]<0)              /* If 1st indicator variable   */
    {                            /*  is negative, V1 is null    */

 .
 .
 .
    }

 .
 .
 .
    strcpy(argv[2],locv2);
                                 /* Assign a value to V2        */
    *(++tempint) = 0;            /* Assign 0 to V2's indicator  */
                                 /*  variable                   */
  }
}
```

Figure 179. An example of GENERAL WITH NULLS linkage in C

```
            CBL RENT
            IDENTIFICATION DIVISION.
            *************************************************************
            * CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
            * GENERAL WITH NULLS LINKAGE CONVENTION.                   *
            *************************************************************
            PROGRAM-ID.   B.

              :
              :
            DATA DIVISION.

              :
              :
            LINKAGE SECTION.
            *************************************************************
            * DECLARE THE PARAMETERS AND THE INDICATOR ARRAY THAT      *
            *  WERE PASSED BY THE SQL STATEMENT CALL HERE.             *
            *************************************************************
            01 V1  PIC S9(9) USAGE COMP.
            01 V2  PIC X(9).
            *
            01 INDARRAY.
                10 INDVAR  PIC S9(4) USAGE COMP OCCURS 2 TIMES.

              :
              :
            PROCEDURE DIVISION USING V1, V2, INDARRAY.
            *************************************************************
            * THE USING PHRASE INDICATES THAT VARIABLES  V1, V2, AND   *
            *  INDARRAY WERE PASSED BY THE CALLING PROGRAM.            *
            *************************************************************

              :
              :
            **************************
            * TEST WHETHER V1 IS NULL *
            **************************
              IF INDARRAY(1) < 0
                PERFORM NULL-PROCESSING.

              :
              :
            ***************************************
            * ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
            *  AND ITS INDICATOR VARIABLE          *
            ***************************************
                MOVE '123456789' TO V2.
                MOVE ZERO TO INDARRAY(2).
```

Figure 180. An example of GENERAL WITH NULLS linkage in COBOL

```
*PROCESS SYSTEM(MVS);
A: PROC(V1, V2, INDSTRUC) OPTIONS(MAIN NOEXECOPS REENTRANT);
 /****************************************************************/
 /* Code for a PL/I language stored procedure that uses the     */
 /* GENERAL WITH NULLS linkage convention.                      */
 /****************************************************************/
 /****************************************************************/
 /* Indicate on the PROCEDURE statement that two parameters     */
 /*  and an indicator variable structure were passed by the SQL */
 /*  statement CALL. Then declare them below.                   */
 /* For PL/I, you must declare an indicator variable structure, */
 /*  not an array.                                              */
 /****************************************************************/
  DCL V1 BIN FIXED(31),
      V2 CHAR(9);
  DCL
     01 INDSTRUC,
        02 IND1 BIN FIXED(15),
        02 IND2 BIN FIXED(15);

  .
  .
  IF IND1 < 0 THEN
   CALL NULLVAL;        /* If indicator variable is negative   */
                        /*  then V1 is null                    */

  .
  .
  V2 = '123456789';    /* Assign a value to output variable V2 */
  IND2 = 0;            /* Assign 0 to V2's indicator variable  */
```

Figure 181. An example of GENERAL WITH NULLS linkage in PL/I

Example of stored procedure linkage convention DB2SQL

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored
procedure uses the DB2SQL linkage convention to receive parameters. These
examples also show how a stored procedure receives the DBINFO structure.

For these examples, assume that a C application has the following parameter
declarations and CALL statement:

```
/*********************************************************/
/* Parameters for the SQL statement CALL                 */
/*********************************************************/
  long int v1;
  char v2[10];                    /* Allow an extra byte for   */
                                  /*  the null terminator      */
/*********************************************************/
/* Indicator variables                                   */
/*********************************************************/
  short int ind1;
  short int ind2;

   .
   .
  ind1 = 0;                       /* Remember to initialize the */
                                  /*  input parameter's indicator*/
                                  /*  variable before executing  */
                                  /*  the CALL statement          */
  EXEC SQL CALL B (:v1 :indstruc.ind1, :v2 :ind1, :ind2);

   .
   .
```

In the CREATE PROCEDURE statement, the parameters are defined like this:

```
V1 INT IN, V2 CHAR(9) OUT
```

Figure 182, Figure 183, Figure 184, Figure 185, and Figure 186 show how a stored procedure in each language receives these parameters.

```
**********************************************************************
*  CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES      *
*  THE DB2SQL LINKAGE CONVENTION.                                 *
**********************************************************************
B       CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
        USING PROGAREA,R13
**********************************************************************
*  BRING UP THE LANGUAGE ENVIRONMENT.                             *
**********************************************************************

    :
    :
**********************************************************************
*  GET THE PASSED PARAMETER VALUES. THE DB2SQL LINKAGE           *
*  CONVENTION IS AS FOLLOWS:                                      *
*  ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N        *
*  PARAMETERS ARE PASSED, THERE ARE 2N+4 POINTERS. THE FIRST      *
*  N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS      *
*  WITH THE GENERAL LINKAGE CONVENTION. THE NEXT N POINTERS ARE   *
*  THE ADDRESSES OF THE INDICATOR VARIABLE VALUES.  THE LAST      *
*  4 POINTERS (5, IF DBINFO IS PASSED) ARE THE ADDRESSES OF       *
*  INFORMATION ABOUT THE STORED PROCEDURE ENVIRONMENT AND         *
*  EXECUTION RESULTS.                                             *
**********************************************************************
        L     R7,0(R1)          GET POINTER TO V1
        MVC   LOCV1(4),0(R7)    MOVE VALUE INTO LOCAL COPY OF V1
        L     R7,8(R1)          GET POINTER TO 1ST INDICATOR VARIABLE
        MVC   LOCI1(2),0(R7)    MOVE VALUE INTO LOCAL STORAGE
        L     R7,20(R1)         GET POINTER TO STORED PROCEDURE NAME
        MVC   LOCSPNM(20),0(R7) MOVE VALUE INTO LOCAL STORAGE
        L     R7,24(R1)         GET POINTER TO DBINFO
        MVC   LOCDBINF(DBINFLN),0(R7)
*                               MOVE VALUE INTO LOCAL STORAGE
        LH    R7,LOCI1          GET INDICATOR VARIABLE FOR V1
        LTR   R7,R7             CHECK IF IT IS NEGATIVE
        BM    NULLIN            IF SO, V1 IS NULL

    :
    :

        L     R7,4(R1)          GET POINTER TO V2
        MVC   0(9,R7),LOCV2     MOVE A VALUE INTO OUTPUT VAR V2
        L     R7,12(R1)         GET POINTER TO INDICATOR VAR 2
        MVC   0(2,R7),=H'0'     MOVE ZERO TO V2'S INDICATOR VAR
        L     R7,16(R1)         GET POINTER TO SQLSTATE
        MVC   0(5,R7),=CL5'xxxxx' MOVE xxxxx TO SQLSTATE

    :
    :
        CEETERM  RC=0
```

Figure 182. An example of DB2SQL linkage in assembler (Part 1 of 2)

```
      ****************************************************************
      *  VARIABLE DECLARATIONS AND EQUATES                          *
      ****************************************************************
      R1       EQU   1                REGISTER 1
      R7       EQU   7                REGISTER 7
      PPA      CEEPPA ,               CONSTANTS DESCRIBING THE CODE BLOCK
               LTORG ,                PLACE LITERAL POOL HERE
      PROGAREA DSECT
               ORG   *+CEEDSASZ       LEAVE SPACE FOR DSA FIXED PART
      LOCV1    DS    F                LOCAL COPY OF PARAMETER V1
      LOCV2    DS    CL9              LOCAL COPY OF PARAMETER V2
      LOCI1    DS    H                LOCAL COPY OF INDICATOR 1
      LOCI2    DS    H                LOCAL COPY OF INDICATOR 2
      LOCSQST  DS    CL5              LOCAL COPY OF SQLSTATE
      LOCSPNM  DS    H,CL27           LOCAL COPY OF STORED PROC NAME
      LOCSPSNM DS    H,CL18           LOCAL COPY OF SPECIFIC NAME
      LOCDIAG  DS    H,CL70           LOCAL COPY OF DIAGNOSTIC DATA
      LOCDBINF DS    0H               LOCAL COPY OF DBINFO DATA
      DBNAMELN DS    H                DATABASE NAME LENGTH
      DBNAME   DS    CL128            DATABASE NAME
      AUTHIDLN DS    H                APPL AUTH ID LENGTH
      AUTHID   DS    CL128            APPL AUTH ID
      ASC_SBCS DS    F                ASCII SBCS CCSID
      ASC_DBCS DS    F                ASCII DBCS CCSID
      ASC_MIXD DS    F                ASCII MIXED CCSID
      EBC_SBCS DS    F                EBCDIC SBCS CCSID
      EBC_DBCS DS    F                EBCDIC DBCS CCSID
      EBC_MIXD DS    F                EBCDIC MIXED CCSID
      ENCODE   DS    F                PROCEDURE ENCODING SCHEME
    | UNI_SBCS DS    F                UNICODE SBCS CCSID
    | UNI_DBCS DS    F                UNICODE DBCS CCSID
    | UNI_MIXD DS    F                UNICODE MIXED CCSID
    | RESERV0  DS    CL8              RESERVED
      TBQUALLN DS    H                TABLE QUALIFIER LENGTH
      TBQUAL   DS    CL128            TABLE QUALIFIER
      TBNAMELN DS    H                TABLE NAME LENGTH
      TBNAME   DS    CL128            TABLE NAME
      CLNAMELN DS    H                COLUMN NAME LENGTH
      COLNAME  DS    CL128            COLUMN NAME
      RELVER   DS    CL8              DBMS RELEASE AND VERSION
      PLATFORM DS    F                DBMS OPERATING SYSTEM
      NUMTFCOL DS    H                NUMBER OF TABLE FUNCTION COLS USED
      RESERV1  DS    CL24             RESERVED
      TFCOLNUM DS    A                POINTER TO TABLE FUNCTION COL LIST
      APPLID   DS    A                POINTER TO APPLICATION ID
      RESERV2  DS    CL20             RESERVED
      DBINFLN  EQU   *-LOCDBINF       LENGTH OF DBINFO

          .
          .
          .
      PROGSIZE EQU   *-PROGAREA
               CEEDSA ,               MAPPING OF THE DYNAMIC SAVE AREA
               CEECAA ,               MAPPING OF THE COMMON ANCHOR AREA
               END   B
```

Figure 182. An example of DB2SQL linkage in assembler (Part 2 of 2)

```
                 #pragma runopts(plist(os))
                 #include <;stdlib.h>
                 #include <;stdio.h>

                 main(argc,argv)
                   int argc;
                   char *argv[];
                 {
                   int parm1;
                   short int ind1;
                   char p_proc[28];
                   char p_spec[19];
                   /**************************************************/
                   /* Assume that the SQL CALL statment included     */
                   /* 3 input/output parameters in the parameter list.*/
                   /* The argv vector will contain these entries:    */
                   /*      argv[0]        1    contains load module  */
                   /*      argv[1-3]      3    input/output parms     */
                   /*      argv[4-6]      3    null indicators        */
                   /*      argv[7]        1    SQLSTATE variable      */
                   /*      argv[8]        1    qualified proc name    */
                   /*      argv[9]        1    specific proc name     */
                   /*      argv[10]       1    diagnostic string      */
                   /*      argv[11]      + 1   dbinfo                 */
                   /*                   ------                        */
                   /*                    12    for the argc variable */
                   /**************************************************/
                   if argc<>12 {
              .
              .
              .
                   /* We end up here when invoked with wrong number of parms */
                   }
```

Figure 183. An example of DB2SQL linkage for a C stored procedure written as a main program (Part 1 of 2)

```
/***************************************************/
/* Assume the first parameter is an integer.       */
/* The code below shows how to copy the integer    */
/* parameter into the application storage.         */
/***************************************************/
parm1 = *(int *) argv[1];
/***************************************************/
/* We can access the null indicator for the first */
/* parameter on the SQL CALL as follows:           */
/***************************************************/
ind1 = *(short int *) argv[4];
/***************************************************/
/* We can use the expression below to assign       */
/* 'xxxxx' to the SQLSTATE returned to caller on   */
/* the SQL CALL statement.                         */
/***************************************************/
strcpy(argv[7],"xxxxx/0");
/***************************************************/
/* We obtain the value of the qualified procedure  */
/* name with this expression.                      */
/***************************************************/
strcpy(p_proc,argv[8]);
/***************************************************/
/* We obtain the value of the specific procedure   */
/* name with this expression.                      */
/***************************************************/
strcpy(p_spec,argv[9]);
/***************************************************/
/* We can use the expression below to assign       */
/* 'yyyyyyyy' to the diagnostic string returned    */
/* in the SQLDA associated with the CALL statement.*/
/***************************************************/
  strcpy(argv[10],"yyyyyyyy/0");
   :
   :
}
```

Figure 183. An example of DB2SQL linkage for a C stored procedure written as a main program (Part 2 of 2)

```
#pragma linkage(myproc,fetchable)
#include <stdlib.h>
#include <stdio.h>
struct sqlsp_dbinfo
{
  unsigned short dbnamelen;   /* database name length      */
  unsigned char  dbname[128]; /* database name             */
  unsigned short authidlen;   /* appl auth id  length      */
  unsigned char  authid[128]; /* appl authorization ID     */
  unsigned long  ascii_sbcs;   /* ASCII SBCS CCSID       */
  unsigned long  ascii_dbcs;   /* ASCII DBCS CCSID       */
  unsigned long  ascii_mixed;  /* ASCII MIXED CCSID      */
  unsigned long  ebcdic_sbcs;  /* EBCDIC SBCS CCSID      */
  unsigned long  ebcdic_dbcs;  /* EBCDIC DBCS CCSID      */
  unsigned long  ebcdic_mixed; /* EBCDIC MIXED CCSID     */
  unsigned long  encode;          /* UDF encode scheme      */
  unsigned long  unicode_sbcs;  /* Unicode SBCS CCSID   */
  unsigned long  unicode_dbcs;  /* Unicode DBCS CCSID    */
  unsigned long  unicode_mixed; /* Unicode MIXED CCSID   */
  unsigned char  reserv0[8];      /* reserved for later use*/
  unsigned short tbqualiflen;     /* table qualifier length */
  unsigned char  tbqualif[128];   /* table qualifer name   */
  unsigned short tbnamelen;       /* table name length      */
  unsigned char  tbname[128];     /* table name             */
  unsigned short colnamelen;      /* column name length     */
  unsigned char  colname[128];    /* column name            */
  unsigned char  relver[8]; /* Database release & version  */
  unsigned long  platform;  /* Database platform           */
  unsigned short numtfcol;  /* # of Tab Fun columns used   */
  unsigned char  reserv1[24];  /* reserved                 */
  unsigned short *tfcolnum;    /* table fn column list     */
  unsigned short *appl_id; /* LUWID for DB2 connection     */
  unsigned char  reserv2[20];  /* reserved                 */
};
```

Figure 184. An example of DB2SQL linkage for a C stored procedure written as a subprogram (Part 1 of 2)

```
void myproc(*parm1 int,                 /* assume INT for PARM1     */
            parm2 char[11],             /* assume CHAR(10) parm2    */
  :
            *p_ind1 short int,          /* null indicator for parm1 */
            *p_ind2 short int,          /* null indicator for parm2 */
  :
            p_sqlstate char[6],         /* SQLSTATE returned to DB2 */
            p_proc char[28],            /* Qualified stored proc name */
            p_spec char[19],            /* Specific stored proc name */
            p_diag char[71],            /* Diagnostic string        */
            struct sqlsp_dbinfo *sp_dbinfo);    /* DBINFO
{
  int l_p1;
  char[11] l_p2;
  short int l_ind1;
  short int l_ind2;
  char[6] l_sqlstate;
  char[28] l_proc;
  char[19] l_spec;
  char[71] l_diag;
  sqlsp_dbinfo *lsp_dbinfo;
  :
  /***************************************************/
  /* Copy each of the parameters in the parameter    */
  /* list into a local variable, just to demonstrate */
  /* how the parameters can be referenced.           */
  /***************************************************/
  l_p1 = *parm1;

  strcpy(l_p2,parm2);

  l_ind1 = *p_ind1;

  l_ind1 = *p_ind2;

  strcpy(l_sqlstate,p_sqlstate);

  strcpy(l_proc,p_proc);

  strcpy(l_spec,p_spec);

  strcpy(l_diag,p_diag);
  memcpy(&lsp_dbinfo,sp_dbinfo,sizeof(lsp_dbinfo));
  :
}
```

Figure 184. An example of DB2SQL linkage for a C stored procedure written as a subprogram (Part 2 of 2)

```
                    CBL RENT
                    IDENTIFICATION DIVISION.
                    .
                    .
                    DATA DIVISION.
                    .
                    .
                    LINKAGE SECTION.
             * Declare each of the input parameters
                    01 PARM1 ...
                    01 PARM2 ...
                    .
                    .
             * Declare a null indicator for each input parameter
                    01 P-IND1 PIC S9(4) USAGE COMP.
                    01 P-IND2 PIC S9(4) USAGE COMP.
                    .
                    .
             * Declare the SQLSTATE that can be set by stored proc
                    01 P-SQLSTATE PIC X(5).
             * Declare the qualified procedure name
                    01 P-PROC.
                       49 P-PROC-LEN PIC 9(4) USAGE BINARY.
                       49 P-PROC-TEXT PIC X(27).
             * Declare the specific procedure name
                    01 P-SPEC.
                       49 P-SPEC-LEN PIC 9(4) USAGE BINARY.
                       49 P-SPEC-TEXT PIC X(18).
             * Declare SQL diagnostic message token
                    01 P-DIAG.
                       49 P-DIAG-LEN PIC 9(4) USAGE BINARY.
                       49 P-DIAG-TEXT PIC X(70).
             ********************************************************
             * Declare the DBINFO structure
             ********************************************************
                    01  SP-DBINFO.
             *        Location length and name
                        02 UDF-DBINFO-LOCATION.
                           49 UDF-DBINFO-LLEN PIC 9(4) USAGE BINARY.
                           49 UDF-DBINFO-LOC  PIC X(128).
             *        Authorization ID length and name
                        02 UDF-DBINFO-AUTHORIZATION.
                           49 UDF-DBINFO-ALEN PIC 9(4) USAGE BINARY.
                           49 UDF-DBINFO-AUTH PIC X(128).
             *        CCSIDs for DB2 for OS/390
                        02 UDF-DBINFO-CCSID PIC X(48).
                        02 UDF-DBINFO-CCSID-REDEFINE REDEFINES UDF-DBINFO-CCSID.
                           03 UDF-DBINFO-ASBCS   PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-ADBCS   PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-AMIXED  PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-ESBCS   PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-EDBCS   PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-EMIXED  PIC 9(9) USAGE BINARY.
                           03 UDF-DBINFO-ENCODE  PIC 9(9) USAGE BINARY.
   |                       03 UDF-DBINFO-USBCS   PIC 9(9) USAGE BINARY.
   |                       03 UDF-DBINFO-UDBCS   PIC 9(9) USAGE BINARY.
   |                       03 UDF-DBINFO-UMIXED  PIC 9(9) USAGE BINARY.
   |                       03 UDF-DBINFO-RESERV0 PIC X(8).
```

Figure 185. An example of DB2SQL linkage in COBOL (Part 1 of 2)

```
*        Schema length and name
     02 UDF-DBINFO-SCHEMA0.
        49 UDF-DBINFO-SLEN PIC 9(4) USAGE BINARY.
        49 UDF-DBINFO-SCHEMA PIC X(128).
*        Table length and name
     02 UDF-DBINFO-TABLE0.
        49 UDF-DBINFO-TLEN PIC 9(4) USAGE BINARY.
        49 UDF-DBINFO-TABLE  PIC X(128).
*        Column length and name
     02 UDF-DBINFO-COLUMN0.
        49 UDF-DBINFO-CLEN PIC 9(4) USAGE BINARY.
        49 UDF-DBINFO-COLUMN  PIC X(128).
*        DB2 release level
     02 UDF-DBINFO-VERREL PIC X(8).
*        Unused
     02 FILLER            PIC X(2).
   *        Database Platform
     02 UDF-DBINFO-PLATFORM PIC 9(9) USAGE BINARY.
   *        # of entries in Table Function column list
     02 UDF-DBINFO-NUMTFCOL PIC 9(4) USAGE BINARY.
   *        reserved
     02 UDF-DBINFO-RESERV1 PIC X(24).
   *        Unused
     02 FILLER            PIC X(2).
   *        Pointer to Table Function column list
     02 UDF-DBINFO-TFCOLUMN PIC 9(9) USAGE BINARY.
   *        Pointer to Application ID
     02 UDF-DBINFO-APPLID  PIC 9(9) USAGE BINARY.
   *        reserved
     02 UDF-DBINFO-RESERV2 PIC X(20).
   *
:
:
 PROCEDURE DIVISION USING PARM1, PARM2,
                          P-IND1, P-IND2,
                          P-SQLSTATE, P-PROC, P-SPEC, P-DIAG,
                          SP-DBINFO.
:
:
```

Figure 185. An example of DB2SQL linkage in COBOL (Part 2 of 2)

```
*PROCESS SYSTEM(MVS);
 MYMAIN: PROC(PARM1, PARM2, ...,
              P_IND1, P_IND2, ...,
              P_SQLSTATE, P_PROC, P_SPEC, P_DIAG, SP_DBINFO)
          OPTIONS(MAIN NOEXECOPS REENTRANT);

 DCL PARM1 ...            /* first parameter  */
 DCL PARM2 ...            /* second parameter */
:
 DCL P_IND1 BIN FIXED(15);/* indicator for 1st parm    */
 DCL P_IND2 BIN FIXED(15);/* indicator for 2nd parm    */
:
 DCL P_SQLSTATE CHAR(5);  /* SQLSTATE to return to DB2 */
 DCL 01 P-PROC  CHAR(27)  /* Qualified procedure name   */
               VARYING;
 DCL 01 P-SPEC  CHAR(18)  /* Specific stored proc       */
               VARYING;
 DCL 01 P-DIAG  CHAR(70)  /* Diagnostic string          */
               VARYING;
 DCL DBINFO PTR;
 DCL 01 SP_DBINFO BASED(DBINFO),          /* Dbinfo           */
       03 UDF_DBINFO_LLEN BIN FIXED(15),  /* location length  */
       03 UDF_DBINFO_LOC  CHAR(128),      /* location name    */
       03 UDF_DBINFO_ALEN BIN FIXED(15),  /* auth ID  length  */
       03 UDF_DBINFO_AUTH CHAR(128),      /* authorization ID */
       03 UDF_DBINFO_CCSID,          /* CCSIDs for DB2 for OS/390*/
        05 R1              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_ASBCS  BIN FIXED(15), /* ASCII SBCS CCSID */
        05 R2              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_ADBCS  BIN FIXED(15), /* ASCII DBCS CCSID */
        05 R3              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_AMIXED BIN FIXED(15), /* ASCII MIXED CCSID */
        05 R4              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_ESBCS  BIN FIXED(15), /* EBCDIC SBCS CCSID */
        05 R5              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_EDBCS  BIN FIXED(15), /* EBCDIC DBCS CCSID */
        05 R6              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_EMIXED BIN FIXED(15), /* EBCDIC MIXED CCSID*/
        05 UDF_DBINFO_ENCODE BIN FIXED(31), /* SP encode scheme  */
        05 R7              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_USBCS  BIN FIXED(15), /* Unicode SBCS CCSID */
        05 R8              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_UDBCS  BIN FIXED(15), /* Unicode DBCS CCSID */
        05 R9              BIN FIXED(15), /* Reserved        */
        05 UDF_DBINFO_UMIXED BIN FIXED(15), /* Unicode MIXED CCSID*/
        05 UDF_DBINFO_RESERV0 CHAR(8),      /* reserved        */
```

Figure 186. An example of DB2SQL linkage in PL/I (Part 1 of 2)

```
              03 UDF_DBINFO_SLEN BIN FIXED(15),     /* schema length    */
              03 UDF_DBINFO_SCHEMA CHAR(128),       /* schema name      */
              03 UDF_DBINFO_TLEN BIN FIXED(15),     /* table length     */
              03 UDF_DBINFO_TABLE  CHAR(128),       /* table name       */
              03 UDF_DBINFO_CLEN BIN FIXED(15),     /* column length    */
              03 UDF_DBINFO_COLUMN  CHAR(128),      /* column name      */
              03 UDF_DBINFO_RELVER  CHAR(8),        /* DB2 release level */
              03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform*/
              03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF cols used*/
              03 UDF_DBINFO_RESERV1 CHAR(24),       /* reserved         */
              03 UDF_DBINFO_TFCOLUMN PTR,      /* -> table fun col list */
              03 UDF_DBINFO_APPLID   PTR,      /* -> application id     */
              03 UDF_DBINFO_RESERV2 CHAR(20);       /* reserved         */
       :
       :
```

Figure 186. An example of DB2SQL linkage in PL/I (Part 2 of 2)

Special considerations for C

In order for the linkage conventions to work correctly when a C language stored procedure runs on MVS, you must include

```
#pragma runopts(PLIST(OS))
```

in your source code. This option is not applicable to other platforms, however. If you plan to use a C stored procedure on other platforms besides MVS, use conditional compilation, as shown in Figure 187, to include this option only when you compile on MVS.

```
#ifdef MVS
 #pragma runopts(PLIST(OS))
#endif

            -- or --

#ifndef WKSTN
 #pragma runopts(PLIST(OS))
#endif
```

Figure 187. Using conditional compilation to include or exclude a statement

Special considerations for PL/I

In order for the linkage conventions to work correctly when a PL/I language stored procedure runs on MVS, you must do the following:
• Include the run-time option NOEXECOPS in your source code.
• Specify the compile-time option SYSTEM(MVS).

For information on specifying PL/I compile-time and run-time options, see *IBM PL/I MVS & VM Programming Guide*.

Using indicator variables to speed processing

If any of your output parameters occupy a great deal of storage, it is wasteful to pass the entire storage areas to your stored procedure. You can use indicator variables in the program that calls the stored procedure to pass only a two byte area to the stored procedure and receive the entire area from the stored procedure. To accomplish this, declare an indicator variable for every large output parameter in your SQL statement CALL. (If you are using the GENERAL WITH NULLS or DB2SQL linkage convention, you must declare indicator variables for all of your

parameters, so you do not need to declare another indicator variable.) Assign a negative value to each indicator variable associated with a large output variable. Then include the indicator variables in the CALL statement. This technique can be used whether the stored procedure linkage convention is GENERAL, GENERAL WITH NULLS, or DB2SQL.

For example, suppose that a stored procedure that is defined with the GENERAL linkage convention takes one integer input parameter and one character output parameter of length 6000. You do not want to pass the 6000 byte storage area to the stored procedure. A PL/I program containing these statements passes only two bytes to the stored procedure for the output variable and receives all 6000 bytes from the stored procedure:

```
DCL INTVAR BIN FIXED(31);      /* This is the input variable   */
DCL BIGVAR(6000);              /* This is the output variable  */
DCL I1 BIN FIXED(15);          /* This is an indicator variable */
    ⋮
I1 = -1;                       /* Setting I1 to -1 causes only  */
                               /*  a two byte area representing */
                               /*  I1 to be passed to the       */
                               /*  stored procedure, instead of */
                               /*  the 6000 byte area for BIGVAR*/
EXEC SQL CALL PROCX(:INTVAR, :BIGVAR INDICATOR :I1);
```

Declaring data types for passed parameters

A stored procedure in any language except REXX must declare each parameter passed to it. In addition, the stored procedure definition must contain a compatible SQL data type declaration for each parameter. For information on creating a stored procedure definition, see "Defining your stored procedure to DB2" on page 531.

For languages other than REXX: For all data types except LOBs, ROWIDs, and locators, see the tables listed in Table 62 for the host data types that are compatible with the data types in the stored procedure definition. For LOBs, ROWIDs, and locators, see tables Table 63, Table 64 on page 595, Table 65 on page 596, and Table 66 on page 597.

For REXX: See "Calling a stored procedure from a REXX Procedure" on page 605 for information on DB2 data types and corresponding parameter formats.

Table 62. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	Table 8 on page 115
C	Table 10 on page 133
COBOL	Table 13 on page 156
PL/I	Table 17 on page 184

Table 63. Compatible assembler language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	Assembler declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	DS FL4

Table 63. Compatible assembler language declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	Assembler declaration
BLOB(*n*)	If n <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CLn If n > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(*n*-65535)
CLOB(*n*)	If n <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CLn If n > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(*n*-65535)
DBCLOB(*n*)	If m (=2*n) <= 65534: var DS 0FL4 var_length DS FL4 var_data DS CLm If m > 65534: var DS 0FL4 var_length DS FL4 var_data DS CL65534 ORG var_data+(*m*-65534)
ROWID	DS HL2,CL40

Table 64. Compatible C language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	C declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	unsigned long
BLOB(*n*)	struct {unsigned long length; char data[n]; } var;
CLOB(*n*)	struct {unsigned long length; char var_data[n]; } var;
DBCLOB(*n*)	struct {unsigned long length; wchar_t data[n]; } var;

ROWID	struct {
	    short int length;
	    char data[40];
	} var;

Table 65. Compatible COBOL declarations for LOBs, ROWIDs, and locators

SQL data type in definition	COBOL declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	01 var PIC S9(9) USAGE IS BINARY.

BLOB(*n*)	If n <= 32767:
	01 var.
	    49 var-LENGTH PIC 9(9)
	         USAGE COMP.
	    49 var-DATA PIC X(*n*).
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	         USAGE COMP.
	  02 var-DATA.
	    49 FILLER
	      PIC X(32767).
	    49 FILLER
	      PIC X(32767).
	    ⋮
	    49 FILLER
	      PIC X(mod(*n*,32767)).

CLOB(*n*)	If n <= 32767:
	01 var.
	    49 var-LENGTH PIC 9(9)
	         USAGE COMP.
	    49 var-DATA PIC X(*n*).
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	         USAGE COMP.
	  02 var-DATA.
	    49 FILLER
	      PIC X(32767).
	    49 FILLER
	      PIC X(32767).
	    ⋮
	    49 FILLER
	      PIC X(mod(*n*,32767)).

Table 65. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration

DBCLOB(*n*)	If n <= 32767:
	01 var.
	    49 var-LENGTH PIC 9(9)
	        USAGE COMP.
	    49 var-DATA PIC G(*n*)
	      USAGE DISPLAY-1.
	If length > 32767:
	01 var.
	  02 var-LENGTH PIC S9(9)
	        USAGE COMP.
	  02 var-DATA.
	    49 FILLER
	      PIC G(32767)
	   USAGE DISPLAY-1.
	    49 FILLER
	      PIC G(32767).
	   USAGE DISPLAY-1.
	     ⋮
	    49 FILLER
	      PIC G(mod(*n*,32767))
	      USAGE DISPLAY-1.

ROWID	01 var.
	  49 var-LEN PIC 9(4)
	      USAGE COMP.
	  49 var-DATA PIC X(40).

Table 66. Compatible PL/I declarations for LOBs, ROWIDs, and locators

SQL data type in definition	PL/I
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	BIN FIXED(31)

BLOB(*n*)	If n <= 32767:
	01 var,
	    03 var_LENGTH
	      BIN FIXED(31),
	    03 var_DATA
	      CHAR(*n*);
	If n > 32767:
	01 var,
	  02 var_LENGTH
	    BIN FIXED(31),
	  02 var_DATA,
	    03 var_DATA1(*n*)
	      CHAR(32767),
	    03 var_DATA2
	      CHAR(mod(*n*,32767));

Table 66. Compatible PL/I declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	PL/I

CLOB(*n*)	If n <= 32767:
	01 var,
	      03 var_LENGTH
	         BIN FIXED(31),
	      03 var_DATA
	         CHAR(*n*);
	If n > 32767:
	01 var,
	    02 var_LENGTH
	       BIN FIXED(31),
	    02 var_DATA,
	       03 var_DATA1(*n*)
	          CHAR(32767),
	       03 var_DATA2
	          CHAR(mod(*n*,32767));

DBCLOB(*n*)	If n <= 16383:
	01 var,
	      03 var_LENGTH
	         BIN FIXED(31),
	      03 var_DATA
	         GRAPHIC(*n*);
	If n > 16383:
	01 var,
	    02 var_LENGTH
	       BIN FIXED(31),
	    02 var_DATA,
	       03 var_DATA1(*n*)
	          GRAPHIC(16383),
	       03 var_DATA2
	          GRAPHIC(mod(*n*,16383));

ROWID	CHAR(40) VAR

Tables of results: Each high-level language definition for stored procedure parameters supports only a single instance (a scalar value) of the parameter. There is no support for structure, array, or vector parameters. Because of this, the SQL statement CALL limits the ability of an application to return some kinds of tables. For example, an application might need to return a table that represents multiple occurrences of one or more of the parameters passed to the stored procedure. Because the SQL statement CALL cannot return more than one set of parameters, use one of the following techniques to return such a table:

- Put the data that the application returns in a DB2 table. The calling program can receive the data in one of these ways:
  - The calling program can fetch the rows from the table directly. Specify FOR FETCH ONLY or FOR READ ONLY on the SELECT statement that retrieves data from the table. A block fetch can retrieve the required data efficiently.
  - The stored procedure can return the contents of the table as a result set. See "Writing a stored procedure to return result sets to a DRDA client" on page 545 and "Writing a DB2 for OS/390 and z/OS client program or SQL procedure to receive result sets" on page 599 for more information.
- Convert tabular data to string format and return it as a character string parameter to the calling program. The calling program and the stored procedure can establish a convention for interpreting the content of the character string. For

example, the SQL statement CALL can pass a 1920-byte character string parameter to a stored procedure, allowing the stored procedure to return a 24x80 screen image to the calling program.

Writing a DB2 for OS/390 and z/OS client program or SQL procedure to receive result sets

You can write a program to receive result sets in one of two ways:

- For a fixed number of result sets, for which you know the contents

  This is the only way in which you can write an SQL procedure to return result sets.

- For a variable number of result sets, for which you do not know the contents

The first alternative is simpler to write, but if you use the second alternative, you do not need to make major modifications to your client program if the stored procedure changes.

There are seven basic steps for receiving result sets:

1. Declare a locator variable for each result set that will be returned.

   If you do not know how many result sets will be returned, declare enough result set locators for the maximum number of result sets that might be returned.

2. Call the stored procedure and check the SQL return code.

   If the SQLCODE from the CALL statement is +466, the stored procedure has returned result sets.

3. Determine how many result sets the stored procedure is returning.

   If you already know how many result sets the stored procedure returns, you can skip this step.

   Use the SQL statement DESCRIBE PROCEDURE to determine the number of result sets. DESCRIBE PROCEDURE places information about the result sets in an SQLDA. Make this SQLDA large enough to hold the maximum number of result sets that the stored procedure might return. When the DESCRIBE PROCEDURE statement completes, the fields in the SQLDA contain the following values:

   - SQLD contains the number of result sets returned by the stored procedure.
   - Each SQLVAR entry gives information about a result set. In an SQLVAR entry:
     - The SQLNAME field contains the name of the SQL cursor used by the stored procedure to return the result set.
     - The SQLIND field contains the value -1. This indicates that no estimate of the number of rows in the result set is available.
     - The SQLDATA field contains the value of the result set locator, which is the address of the result set.

4. Link result set locators to result sets.

   You can use the SQL statement ASSOCIATE LOCATORS to link result set locators to result sets. The ASSOCIATE LOCATORS statement assigns values to the result set locator variables. If you specify more locators than the number of result sets returned, DB2 ignores the extra locators.

   To use the ASSOCIATE LOCATORS statement, you must embed it in an application or SQL procedure. You cannot execute ASSOCIATE LOCATORS dynamically.

   If you executed the DESCRIBE PROCEDURE statement previously, the result set locator values are in the SQLDATA fields of the SQLDA. You can copy the

values from the SQLDATA fields to the result set locators manually, or you can execute the ASSOCIATE LOCATORS statement to do it for you.

The stored procedure name that you specify in an ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statement must match the stored procedure name in the CALL statement that returns the result sets. That is:

- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is unqualified, the stored procedure name in the CALL statement must be unqualified.
- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is qualified with a schema name, the stored procedure name in the CALL statement must be qualified with a schema name.
- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is qualified with a location name and a schema name, the stored procedure name in the CALL statement must be qualified with a location name and a schema name.

5. Allocate cursors for fetching rows from the result sets.

   Use the SQL statement ALLOCATE CURSOR to link each result set with a cursor. Execute one ALLOCATE CURSOR statement for each result set. The cursor names can be different from the cursor names in the stored procedure.

   To use the ALLOCATE CURSOR statement, you must embed it in an application or SQL procedure. You cannot execute ALLOCATE CURSOR dynamically.

6. Determine the contents of the result sets.

   If you already know the format of the result set, you can skip this step.

   Use the SQL statement DESCRIBE CURSOR to determine the format of a result set and put this information in an SQLDA. For each result set, you need an SQLDA big enough to hold descriptions of all columns in the result set.

   You can use DESCRIBE CURSOR only for cursors for which you executed ALLOCATE CURSOR previously.

   After you execute DESCRIBE CURSOR, if the cursor for the result set is declared WITH HOLD, the high-order bit of the eighth byte of field SQLDAID in the SQLDA is set to 1.

7. Fetch rows from the result sets into host variables by using the cursors that you allocated with the ALLOCATE CURSOR statements.

   If you executed the DESCRIBE CURSOR statement, perform these steps before you fetch the rows:

   a. Allocate storage for host variables and indicator variables. Use the contents of the SQLDA from the DESCRIBE CURSOR statement to determine how much storage you need for each host variable.

   b. Put the address of the storage for each host variable in the appropriate SQLDATA field of the SQLDA.

   c. Put the address of the storage for each indicator variable in the appropriate SQLIND field of the SQLDA.

   Fetching rows from a result set is the same as fetching rows from a table.

You do not need to connect to the remote location when you execute these statements:
- DESCRIBE PROCEDURE
- ASSOCIATE LOCATORS
- ALLOCATE CURSOR
- DESCRIBE CURSOR

- FETCH
- CLOSE

For the syntax of result set locators in each host language, see "Chapter 9. Embedding SQL statements in host languages" on page 107. For the syntax of result set locators in SQL procedures, see Chapter 6 of *DB2 SQL Reference*. For the syntax of the ASSOCIATE LOCATORS, DESCRIBE PROCEDURE, ALLOCATE CURSOR, and DESCRIBE CURSOR statements, see Chapter 5 of *DB2 SQL Reference*.

Figure 188 on page 602 and Figure 189 on page 603 show C language code that accomplishes each of these steps. Coding for other languages is similar. For a more complete example of a C language program that receives result sets, see "Examples of using stored procedures" on page 890.

Figure 188 on page 602 demonstrates how you receive result sets when you know how many result sets are returned and what is in each result set.

```
/*************************************************************/
/* Declare result set locators.  For this example,         */
/* assume you know that two result sets will be returned.   */
/* Also, assume that you know the format of each result set. */
/*************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
   static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2;
  EXEC SQL END DECLARE SECTION;
  .
  .
  .
/*************************************************************/
/* Call stored procedure P1.                                */
/* Check for SQLCODE +466, which indicates that result sets */
/* were returned.                                           */
/*************************************************************/
  EXEC SQL CALL P1(:parm1, :parm2, ...);
  if(SQLCODE==+466)
  {
  /*************************************************************/
  /* Establish a link between each result set and its         */
  /* locator using the ASSOCIATE LOCATORS.                    */
  /*************************************************************/
    EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2) WITH PROCEDURE P1;
  .
  .
  .
  /*************************************************************/
  /* Associate a cursor with each result set.                 */
  /*************************************************************/
    EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
    EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
  /*************************************************************/
  /* Fetch the result set rows into host variables.           */
  /*************************************************************/
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C1 INTO :order_no, :cust_no;
  .
  .
  .
    }
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C2 :order_no, :item_no, :quantity;
  .
  .
  .
    }
  }
```

Figure 188. Receiving known result sets

Figure 189 on page 603 demonstrates how you receive result sets when you do not know how many result sets are returned or what is in each result set.

```
/**************************************************************/
/* Declare result set locators.  For this example,           */
/* assume that no more than three result sets will be        */
/* returned, so declare three locators.  Also, assume        */
/* that you do not know the format of the result sets.       */
/**************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
  EXEC SQL END DECLARE SECTION;

  .
  .
  .
```

Figure 189. Receiving unknown result sets (Part 1 of 3)

```
/**************************************************************/
/* Call stored procedure P2.                                 */
/* Check for SQLCODE +466, which indicates that result sets  */
/* were returned.                                            */
/**************************************************************/
  EXEC SQL CALL P2(:parm1, :parm2, ...);
  if(SQLCODE==+466)
  {
  /**************************************************************/
  /* Determine how many result sets P2 returned, using the     */
  /* statement DESCRIBE PROCEDURE.  :proc_da is an SQLDA        */
  /* with enough storage to accommodate up to three SQLVAR     */
  /* entries.                                                  */
  /**************************************************************/
    EXEC SQL DESCRIBE PROCEDURE P2 INTO :proc_da;

  .
  .
  .
  /**************************************************************/
  /* Now that you know how many result sets were returned,     */
  /* establish a link between each result set and its          */
  /* locator using the ASSOCIATE LOCATORS.  For this example,  */
  /* we assume that three result sets are returned.            */
  /**************************************************************/
    EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P2;

  .
  .
  .
  /**************************************************************/
  /* Associate a cursor with each result set.                  */
  /**************************************************************/
    EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
    EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
    EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :loc3;
```

Figure 189. Receiving unknown result sets (Part 2 of 3)

```
/**************************************************************/
/* Use the statement DESCRIBE CURSOR to determine the        */
/* format of each result set.                                */
/**************************************************************/
  EXEC SQL DESCRIBE CURSOR C1 INTO :res_da1;
  EXEC SQL DESCRIBE CURSOR C2 INTO :res_da2;
  EXEC SQL DESCRIBE CURSOR C3 INTO :res_da3;

  ⋮

/**************************************************************/
/* Assign values to the SQLDATA and SQLIND fields of the     */
/* SQLDAs that you used in the DESCRIBE CURSOR statements.    */
/* These values are the addresses of the host variables and  */
/* indicator variables into which DB2 will put result set    */
/* rows.                                                      */
/**************************************************************/

  ⋮

/**************************************************************/
/* Fetch the result set rows into the storage areas          */
/* that the SQLDAs point to.                                 */
/**************************************************************/
  while(SQLCODE==0)
  {
    EXEC SQL FETCH C1 USING :res_da1;

  ⋮

  }
  while(SQLCODE==0)
  {
    EXEC SQL FETCH C2 USING :res_da2;

  ⋮

  }
  while(SQLCODE==0)
  {
    EXEC SQL FETCH C3 USING :res_da3;

  ⋮

  }
 }
```

Figure 189. Receiving unknown result sets (Part 3 of 3)

Figure 190 on page 605 demonstrates how you can use an SQL procedure to receive result sets.

```
                    DECLARE RESULT1 RESULT_SET_LOCATOR VARYING;
                    DECLARE RESULT2 RESULT_SET_LOCATOR VARYING;
                      ⋮
                    CALL TARGETPROCEDURE();
                    ASSOCIATE RESULT SET LOCATORS(RESULT1,RESULT2)
                      WITH PROCEDURE TARGETPROCEDURE;
                    ALLOCATE RSCUR1 CURSOR FOR RESULT1;
                    ALLOCATE RSCUR2 CURSOR FOR RESULT2;
                    WHILE AT_END = 0 DO
                      FETCH RSCUR1 INTO VAR1;
                      SET TOTAL1 = TOTAL1 + VAR1;
                    END WHILE;
                    WHILE AT_END = 0 DO
                      FETCH RSCUR2 INTO VAR2;
                      SET TOTAL2 = TOTAL2 + VAR2;
                    END WHILE;
                      ⋮
```

Figure 190. Receiving result sets in an SQL procedure

Accessing transition tables in a stored procedure

When you write a user-defined function, external stored procedure, or SQL procedure that is to be invoked from a trigger, you might need access to transition tables for the trigger. The technique for accessing the transition tables is the same for user-defined functions and stored procedures, and is described in "Accessing transition tables in a user-defined function or stored procedure" on page 278.

Calling a stored procedure from a REXX Procedure

The format of the parameters that you pass in the CALL statement in a REXX procedure must be compatible with the data types of the parameters in the CREATE PROCEDURE statement. Table 67 lists each SQL data type that you can specify for the parameters in the CREATE PROCEDURE statement and the corresponding format for a REXX parameter that represents that data type.

Table 67. Parameter formats for a CALL statement in a REXX procedure

SQL data type	REXX format
SMALLINT INTEGER	A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus or minus sign. This format also applies to indicator variables that are passed as parameters.
DECIMAL(*p,s*) NUMERIC(*p,s*)	A string of numerics that has a decimal point but no exponent identifier. The first character can be a plus or minus sign.
REAL FLOAT(*n*) DOUBLE	A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus or minus sign and a series of numerics).
CHARACTER(*n*) VARCHAR(*n*) VARCHAR(*n*) FOR BIT DATA	A string of length *n*, enclosed in single quotation marks.
GRAPHIC(*n*) VARGRAPHIC(*n*)	The character G followed by a string enclosed in single quotation marks. The string within the quotation marks begins with a shift-out character (X'0E') and ends with a shift-in character (X'0F'). Between the shift-out character and shift-in character are *n* double-byte characters.
DATE	A string of length 10, enclosed in single quotation marks. The format of the string depends on the value of field DATE FORMAT that you specify when you install DB2. See Chapter 2 of *DB2 SQL Reference* for valid date string formats.

Table 67. Parameter formats for a CALL statement in a REXX procedure (continued)

SQL data type	REXX format
TIME	A string of length 8, enclosed in single quotation marks. The format of the string depends on the value of field TIME FORMAT that you specify when you install DB2. See Chapter 2 of *DB2 SQL Reference* for valid time string formats.
TIMESTAMP	A string of length 26, enclosed in single quotation marks. The string has the format *yyyy-mm-dd-hh.mm.ss.nnnnnn*.

Figure 191 on page 607 demonstrates how a REXX procedure calls the stored procedure in Figure 167 on page 550. The REXX procedure performs the following actions:

- Connects to the DB2 subsystem that was specified by the REXX procedure invoker.
- Calls the stored procedure to execute a DB2 command that was specified by the REXX procedure invoker.
- Retrieves rows from a result set that contains the command output messages.

```
/* REXX */
PARSE ARG SSID COMMAND                         /* Get the SSID to connect to */
                                               /* and the DB2 command to be  */
                                               /* executed                   */
  /****************************************************************/
  /*  Set up the host command environment for SQL calls.        */
  /****************************************************************/
"SUBCOM DSNREXX"                               /* Host cmd env available? */
IF RC THEN                                     /* No--make one             */
   S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
  /****************************************************************/
  /*  Connect to the DB2 subsystem.                             */
  /****************************************************************/
ADDRESS DSNREXX "CONNECT" SSID

IF SQLCODE ¬= 0 THEN CALL SQLCA

PROC = 'COMMAND'

RESULTSIZE = 32703
RESULT = LEFT(' ',RESULTSIZE,' ')
  /****************************************************************/
  /*  Call the stored procedure that executes the DB2 command.  */
  /*  The input variable (COMMAND) contains the DB2 command.    */
  /*  The output variable (RESULT) will contain the return area */
  /*  from the IFI COMMAND call after the stored procedure      */
  /*  executes.                                                 */
  /****************************************************************/
ADDRESS DSNREXX "EXECSQL" ,
"CALL" PROC "(:COMMAND, :RESULT)"
IF SQLCODE < 0 THEN CALL SQLCA

SAY 'RETCODE ='RETCODE
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1',',
            || SQLERRD.2',',
            || SQLERRD.3',',
            || SQLERRD.4',',
            || SQLERRD.5',',
            || SQLERRD.6

SAY 'SQLWARN ='SQLWARN.0',',
            || SQLWARN.1',',
            || SQLWARN.2',',
            || SQLWARN.3',',
            || SQLWARN.4',',
            || SQLWARN.5',',
            || SQLWARN.6',',
            || SQLWARN.7',',
            || SQLWARN.8',',
            || SQLWARN.9',',
            || SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
SAY C2X(RESULT) "'"||RESULT||"'"
```

Figure 191. Example of a REXX procedure that calls a stored procedure (Part 1 of 3)

```
           /******************************************************************/
           /*  Display the IFI return area in hexadecimal.                  */
           /******************************************************************/
           OFFSET = 4+1
           TOTLEN = LENGTH(RESULT)
           DO WHILE ( OFFSET <  TOTLEN )
             LEN = C2D(SUBSTR(RESULT,OFFSET,2))
             SAY SUBSTR(RESULT,OFFSET+4,LEN-4-1)
             OFFSET = OFFSET + LEN
           END
           /******************************************************************/
           /*  Get information about result sets returned by the            */
           /*  stored procedure.                                            */
           /******************************************************************/
           ADDRESS DSNREXX "EXECSQL DESCRIBE PROCEDURE :PROC INTO :SQLDA"
           IF SQLCODE ¬= 0 THEN CALL SQLCA

           DO I = 1 TO SQLDA.SQLD
             SAY "SQLDA."I".SQLNAME    ="SQLDA.I.SQLNAME";"
             SAY "SQLDA."I".SQLTYPE    ="SQLDA.I.SQLTYPE";"
             SAY "SQLDA."I".SQLLOCATOR ="SQLDA.I.SQLLOCATOR";"
             SAY "SQLDA."I".SQLESTIMATE="SQLDA.I.SQLESTIMATE";"
           END I
           /******************************************************************/
           /*  Set up a cursor to retrieve the rows from the result         */
           /*  set.                                                         */
           /******************************************************************/
           ADDRESS DSNREXX "EXECSQL ASSOCIATE LOCATOR (:RESULT) WITH PROCEDURE :PROC"
           IF SQLCODE ¬= 0 THEN CALL SQLCA
           SAY RESULT

           ADDRESS DSNREXX "EXECSQL ALLOCATE C101 CURSOR FOR RESULT SET :RESULT"
           IF SQLCODE ¬= 0 THEN CALL SQLCA

           CURSOR = 'C101'
           ADDRESS DSNREXX "EXECSQL DESCRIBE CURSOR :CURSOR INTO :SQLDA"
           IF SQLCODE ¬= 0 THEN CALL SQLCA
           /******************************************************************/
           /*  Retrieve and display the rows from the result set, which     */
           /*  contain the command output message text.                     */
           /******************************************************************/
           DO UNTIL(SQLCODE ¬= 0)
             ADDRESS DSNREXX "EXECSQL FETCH C101 INTO :SEQNO, :TEXT"
             IF SQLCODE = 0 THEN
               DO
                 SAY TEXT
               END
           END
           IF SQLCODE ¬= 0 THEN CALL SQLCA

           ADDRESS DSNREXX "EXECSQL CLOSE C101"
           IF SQLCODE ¬= 0 THEN CALL SQLCA

           ADDRESS DSNREXX "EXECSQL COMMIT"
           IF SQLCODE ¬= 0 THEN CALL SQLCA
```

Figure 191. Example of a REXX procedure that calls a stored procedure (Part 2 of 3)

```
      /**************************************************************/
      /*  Disconnect from the DB2 subsystem.                      */
      /**************************************************************/
ADDRESS DSNREXX "DISCONNECT"
IF SQLCODE ¬= 0 THEN CALL SQLCA
      /**************************************************************/
      /*  Delete the host command environment for SQL.            */
      /**************************************************************/
S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX') /* REMOVE CMD ENV  */

RETURN
      /**************************************************************/
      /*  Routine to display the SQLCA                            */
      /**************************************************************/
SQLCA:
TRACE O
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1',',
            || SQLERRD.2',',
            || SQLERRD.3',',
            || SQLERRD.4',',
            || SQLERRD.5',',
            || SQLERRD.6

SAY 'SQLWARN ='SQLWARN.0',',
            || SQLWARN.1',',
            || SQLWARN.2',',
            || SQLWARN.3',',
            || SQLWARN.4',',
            || SQLWARN.5',',
            || SQLWARN.6',',
            || SQLWARN.7',',
            || SQLWARN.8',',
            || SQLWARN.9',',
            || SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
EXIT
```

Figure 191. Example of a REXX procedure that calls a stored procedure (Part 3 of 3)

Preparing a client program

You must prepare the calling program by precompiling, compiling, and link-editing it on the client system.

Before you can call a stored procedure from your embedded SQL application, you must bind a package for the client program on the remote system. You can use the remote DRDA bind capability on your DRDA client system to bind the package to the remote system.

If you have packages that contain SQL CALL statements that you bound before DB2 Version 6, you can get better performance from those packages if you rebind them in DB2 Version 6 or later. Rebinding lets DB2 obtain some information from the catalog at bind time that it obtained at run time before Version 6. Therefore, after you rebind your packages, they run more efficiently because DB2 can do fewer catalog searches at run time.

For an ODBC or CLI application, the DB2 packages and plan associated with the ODBC driver must be bound to DB2 before you can run your application. For information on building client applications on platforms other than DB2 for OS/390 and z/OS to access stored procedures, see one of these documents:
- *DB2 UDB Application Building Guide*
- *DB2 Universal Database for AS/400 SQL Programming with Host Languages*

An MVS client can bind the DBRM to a remote server by specifying a location name on the command BIND PACKAGE. For example, suppose you want a client program to call a stored procedure at location LOCA. You precompile the program to produce DBRM A. Then you can use the command
```
BIND PACKAGE (LOCA.COLLA) MEMBER(A)
```

to bind DBRM A into package collection COLLA at location LOCA.

The plan for the package resides only at the client system.

Running a stored procedure

Stored procedures run as either main programs or subprograms. "Writing a stored procedure as a main program or subprogram" on page 539 contains information on the requirements for each type of stored procedure.

If a stored procedure runs as a *main program,* before each call, Language Environment reinitializes the storage used by the stored procedure. Program variables for the stored procedure do not persist between calls.

If a stored procedure runs as a *subprogram,* Language Environment does not initialize the storage between calls. Program variables for the stored procedure can persist between calls. However, you should not assume that your program variables are available from one stored procedure call to another because:
- Stored procedures from other users can run in an instance of Language Environment between two executions of your stored procedure.
- Consecutive executions of a stored procedure might run in different stored procedures address spaces.
- The MVS operator might refresh Language Environment between two executions of your stored procedure.

DB2 runs stored procedures under the DB2 thread of the calling application, making the stored procedures part of the caller's unit of work.

If both the client and server application environments support two-phase commit, the coordinator controls updates between the application, the server, and the stored procedures. If either side does not support two-phase commit, updates will fail.

If a stored procedure abnormally terminates:
- The calling program receives an SQL error as notification that the stored procedure failed.
- DB2 places the calling program's unit of work in a must-rollback state.
- If the stored procedure does not handle the abend condition, DB2 refreshes the Language Environment environment to recover the storage that the application uses. In most cases, the Language Environment environment does not need to restart.

- If a data set is allocated to the DD name CEEDUMP in the JCL procedure that starts the stored procedures address space, Language Environment writes a small diagnostic dump to this data set. See your system administrator to obtain the dump information. Refer to "Testing a stored procedure" on page 615 for techniques that you can use to diagnose the problem.

How DB2 determines which version of a stored procedure to run

The combination of the schema name and stored procedure name uniquely identify a stored procedure. If you qualify the stored procedure name when you execute a CALL statement to call a stored procedure, there is only one candidate to run. However, if you do not qualify the stored name, DB2 uses the following method to determine which stored procedure to run:

1. DB2 goes through the list of schema names from the PATH bind option or the CURRENT PATH special register from left to right until it finds a schema name for which there exists a stored procedure definition with the name in the CALL statement. DB2 uses the schema names from the PATH bind option for CALL statements of the form

   ```
   CALL literal
   ```

   For CALL statements of the form

   ```
   CALL host-variable
   ```

   DB2 uses schema names from the CURRENT PATH special register.

2. When DB2 finds a stored procedure definition, DB2 executes that stored procedure if the following conditions are true:

   - The caller is authorized to execute the stored procedure.
   - The stored procedure has the same number of parameters as in the CALL statement.

   If both conditions are not true, DB2 continues to go through the list of schemas until it finds a stored procedure that meets both conditions or reaches the end of the list.

3. If DB2 cannot find a suitable stored procedure, it returns an SQL error code for the CALL statement.

Using a single application program to call different versions of a stored procedure

If you want to use the same application program to call different versions of a stored procedure that have the same load module name, follow these steps:

1. When you define each version of the stored procedure, use the same stored procedure name but different schema names and WLM environments.

2. In the program that invokes the stored procedure, specify the unqualified stored procedure name in the CALL statement.

3. Use the SQL path to indicate which version of the stored procedure that the client program should call. You can choose the SQL path in several ways:

   - If the client program is not an ODBC or JDBC™ application, use one of the following methods:

     – Use the CALL *procedure-name* form of the CALL statement. When you bind plans or packages for the program that calls the stored procedure, bind one plan or package for each version of the stored procedure that you want to call. In the PATH bind option for each plan or package, specify the schema name of the stored procedure that you want to call.

- Use the CALL *host-variable* form of the CALL statement. In the client program, use the SET CURRENT PATH statement to specify the schema name of the stored procedure that you want to call.
- If the client program is an ODBC or JDBC application, choose one of the following methods:
  - Use the SET CURRENT PATH statement to specify the schema name of the stored procedure that you want to call.
  - When you bind the stored procedure packages, specify a different collection for each stored procedure package. In the client program, execute the SET CURRENT PACKAGESET statement to point to the package collection that contains the stored procedure that you want to call.

4. When you run the client program, specify the plan or package with the PATH value that matches the schema name name of the stored procedure that you want to call.

For example, suppose that you want to write one program, PROGY, that calls one of two versions of a stored procedure named PROCX. The load module for both stored procedures is named SUMMOD. Each version of SUMMOD is in a different load library. The stored procedures run in different WLM environments, and the startup JCL for each WLM environment includes a STEPLIB concatenation that specifies the correct load library for the stored procedure module.

First, define the two stored procedures in different schemas and different WLM environments:

```
CREATE PROCEDURE TEST.PROCX(V1 INTEGER IN, CHAR(9) OUT)
  LANGUAGE C
  EXTERNAL NAME SUMMOD
  WLM ENVIRONMENT TESTENV;
CREATE PROCEDURE PROD.PROCX(V1 INTEGER IN, CHAR(9) OUT)
  LANGUAGE C
  EXTERNAL NAME SUMMOD
  WLM ENVIRONMENT PRODENV;
```

When you write CALL statements for PROCX in program PROGY, use the unqualified form of the stored procedure name:

```
CALL PROCX(V1,V2);
```

Bind two plans for PROGY, In one BIND statement, specify PATH(TEST). In the other BIND statement, specify PATH(PROD).

To call TEST.PROCX, execute PROGY with the plan that you bound with PATH(TEST). To call PROD.PROCX, execute PROGY with the plan that you bound with PATH(PROD).

Running multiple stored procedures concurrently

Multiple stored procedures can run concurrently, each under its own MVS task (TCB). The maximum number of stored procedures that can run concurrently in a single address space is set at DB2 installation time, on panel DSNTIPX.

See Part 2 of *DB2 Installation Guide* for more information.

You can override that value in the following ways:
- For WLM-established or DB2-established stored procedures address spaces:

- Specify the NUMTCB parameter when you issue the MVS START command to start stored procedures address spaces.
- Edit the JCL procedures that start stored procedures address spaces, and modify the value of the NUMTCB parameter.
- For WLM-established address spaces, when you set up a WLM application environment, specify the parameter

  `NUMTCB=number-of-TCBs`

  in field Start Parameters of panel Create An Application Environment.

To maximize the number of stored procedures that can run concurrently, use the following guidelines:

- Set REGION size to 0 in startup procedures for the stored procedures address spaces to obtain the largest possible amount of storage below the 16MB line.
- Limit storage required by application programs below the 16MB line by:
  - Link editing programs above the line with AMODE(31) and RMODE(ANY) attributes
  - Using the RENT and DATA(31) compiler options for COBOL programs.
- Limit storage required by IBM Language Environment by using these run-time options:
  - HEAP(,,ANY) to allocate program heap storage above the 16MB line
  - STACK(,,ANY,) to allocate program stack storage above the 16MB line
  - STORAGE(,,,4K) to reduce reserve storage area below the line to 4KB
  - BELOWHEAP(4K,,) to reduce the heap storage below the line to 4KB
  - LIBSTACK(4K,,) to reduce the library stack below the line to 4KB
  - ALL31(ON) to indicate all programs contained in the stored procedure run with AMODE(31) and RMODE(ANY).

  You can list these options in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement, if they are not Language Environment installation defaults. For example, the RUN OPTIONS parameter could specify:

  `H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)`

  For more information on creating a stored procedure definition, see "Defining your stored procedure to DB2" on page 531.

- If you use WLM-established address spaces for your stored procedures, assign stored procedures that behave similarly to the same WLM application environment. When the stored procedures within a WLM environment have substantially different performance characteristics, WLM can have trouble characterizing the workload in the WLM environment. As a result, WLM can create too few or too many address spaces. Both problems can increase response times for stored procedures and other DB2 applications.

  For more information on assigning stored procedures to WLM application environments, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Accessing non-DB2 resources

Applications that run in a stored procedures address space can access any resources available to MVS address spaces, such as VSAM files, flat files, MVS/APPC conversations, and IMS or CICS transactions.

Consider the following when you develop stored procedures that access non-DB2 resources:

- When a stored procedure runs in a DB2-established stored procedures address space, DB2 does not coordinate commit and rollback activity on recoverable resources such as IMS or CICS transactions, and MQI messages. DB2 has no knowledge of, and therefore cannot control, the dependency between a stored procedure and a recoverable resource.

- When a stored procedure runs in a WLM-established stored procedures address space, the stored procedure uses the OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS) for commitment control. When DB2 commits or rolls back work in this environment, DB2 coordinates all updates made to recoverable resources by other OS/390 RRS compliant resource managers in the MVS system.

- When a stored procedure runs in a DB2-established stored procedures address space, MVS is not aware that the stored procedures address space is processing work for DB2. One consequence of this is that MVS accesses RACF-protected resources using the user ID associated with the MVS task (*ssnm*SPAS) for stored procedures, not the user ID of the client.

- When a stored procedure runs in a WLM-established stored procedures address space, DB2 can establish a RACF environment for accessing non-DB2 resources. The authority used when the stored procedure accesses protected MVS resources depends on the value of SECURITY in the stored procedure definition:
  - If the value of SECURITY is DB2, the authorization ID associated with the stored procedures address space is used.
  - If the value of SECURITY is USER, the authorization ID under which the CALL statement is executed is used.
  - If the value of SECURITY is DEFINER, the authorization ID under which the CREATE PROCEDURE statement was executed is used.

- Not all non-DB2 resources can tolerate concurrent access by multiple TCBs in the same address space. You might need to serialize the access within your application.

CICS

Stored procedure applications can access CICS by one of the following methods:
- Message Queue Interface (MQI): for asynchronous execution of CICS transactions
- External CICS interface (EXCI): for synchronous execution of CICS transactions
- Advanced Program-to-Program Communication (APPC), using the Common Programming Interface Communications (CPI Communications) application programming interface

For DB2-established address spaces, a CICS application runs as a separate unit of work from the unit of work under which the stored procedure runs. Consequently, results from CICS processing do not affect the completion of stored procedure processing. For example, a CICS transaction in a stored procedure that rolls back a unit of work does not prevent the stored procedure from committing the DB2 unit of work. Similarly, a rollback of the DB2 unit of work does not undo the successful commit of a CICS transaction.

For WLM-established address spaces, if your system is running a release of CICS that uses OS/390 RRS, OS/390 RRS controls commitment of all resources.

IMS

If your system is not running a release of IMS that uses OS/390 RRS, you can use one of the following methods to access DL/I data from your stored procedure:
- Use the CICS EXCI interface to run a CICS transaction synchronously. That CICS transaction can, in turn, access DL/I data.
- Invoke IMS transactions asynchronously using the MQI.
- Use APPC through the CPI Communications application programming interface

---

Testing a stored procedure

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run. Here are some alternative testing strategies to consider.

Debugging the stored procedure as a stand-alone program on a workstation

If you have debugging support on a workstation, you might choose to do most of your development and testing on a workstation, before installing a stored procedure on MVS. This results in very little debugging activity on MVS.

Debugging with the Debug Tool and IBM VisualAge® COBOL

If you have VisualAge COBOL installed on your workstation and the Debug Tool installed on your OS/390 system, you can use the VisualAge COBOL Edit/Compile/Debug component with the Debug Tool to debug a COBOL stored procedure that runs in a WLM-established stored procedures address space. For detailed information on the Debug Tool, see *Debug Tool User's Guide and Reference*.

After you write your COBOL stored procedure and set up the WLM environment, follow these steps to test the stored procedure with the Debug Tool:

1. When you compile the stored procedure, specify the TEST and SOURCE options.

   Ensure that the source listing is stored in a permanent data set. VisualAge COBOL displays that source listing during the debug session.

2. When you define the stored procedure, include run-time option TEST with the suboption VADTCPIP&*ipaddr* in your RUN OPTIONS argument.

   VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge COBOL and is configured for TCP/IP communication with your OS/390 system. *ipaddr* is the IP address of the workstation on which you display your debug information. For example, the RUN OPTIONS value in this stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

```
CREATE PROCEDURE WLMCOB
 (IN INTEGER, INOUT VARCHAR(3000), INOUT INTEGER)
 MODIFIES SQL DATA
 LANGUAGE COBOL EXTERNAL
 PROGRAM TYPE MAIN
 WLM ENVIRONMENT WLMENV1
 RUN OPTIONS 'POSIX(ON),TEST(,,,VADTCPIP&9.63.51.17:*)'
```

3. In the JCL startup procedure for WLM-established stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

```
//DSNWLM   PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//         PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DSN710.SDSNLOAD
//         DD  DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
```

4. On the workstation, start the VisualAge Remote Debugger daemon.

   This daemon waits for incoming requests from TCP/IP.

5. Call the stored procedure.

   When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Debugging an SQL procedure or C language stored procedure with the Debug Tool and C/C++ Productivity Tools for OS/390

If you have the C/C++ Productivity Tools for OS/390 installed on your workstation and the Debug Tool installed on your OS/390 system, you can debug an SQL procedure or C or C++ stored procedure that runs in a WLM-established stored procedures address space. The code against which you run the debug tools is the

C source program that is produced by the program preparation process for the stored procedure. For detailed information on the Debug Tool, see *Debug Tool User's Guide and Reference*.

After you write your C⁺⁺ stored procedure or SQL procedure and set up the WLM environment, follow these steps to test the stored procedure with the Distributed Debugger feature of the C/C⁺⁺ Productivity Tools for OS/390 and the Debug Tool:

1. When you define the stored procedure, include run-time option TEST with the suboption VADTCPIP&*ipaddr* in your RUN OPTIONS argument.

   VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge C⁺⁺ and is configured for TCP/IP communication with your OS/390 system. *ipaddr* is the IP address of the workstation on which you display your debug information. For example, this RUN OPTIONS value in a stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

   ```
   RUN OPTIONS 'POSIX(ON),TEST(,,,VADTCPIP&9.63.51.17:*)'
   ```

2. Precompile the stored procedure.

   Ensure that the modified source program that is the output from the precompile step is in a *permanent, catalogued* data set. For an SQL procedure, the modified C source program that is the output from the *second* precompile step must be in a permament, catalogued data set.

3. Compile the output from the precompile step. Specify the TEST, SOURCE, and OPT(0) compiler options.

4. In the JCL startup procedure for the stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

   ```
   //DSNWLM   PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
   //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
   //         PARM='&DB2SSN,&NUMTCB,&APPLENV'
   //STEPLIB  DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD
   //         DD  DISP=SHR,DSN=CEE.SCEERUN
   //         DD  DISP=SHR,DSN=DSN710.SDSNLOAD
   //         DD  DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
   ```

5. On the workstation, start the Distributed Debugger daemon.

   This daemon waits for incoming requests from TCP/IP.

6. Call the stored procedure.

   When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Debugging with CODE/370

You can use the CoOperative Development Environment/370 licensed program, which works with Language Environment, to test MVS stored procedures written in any of the supported languages. You can use CODE/370 either interactively or in batch mode.

Using CODE/370 interactively: To test a stored procedure interactively using CODE/370, you must use the CODE/370 PWS Debug Tool on a workstation. You must also have CODE/370 installed on the MVS system where the stored procedure runs. To debug your stored procedure using the PWS Debug Tool, do the following:

- Compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.
- Invoke the debug tool. One way to do that is to specify the Language Environment run-time option TEST. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run-time options is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

  For example, if you code this option:

  ```
  TEST(ALL,*,PROMPT,JBJONES%SESSNA:)
  ```

  the parameter values cause the following things to happen:

  ALL
  > The Debug Tool gains control when an attention interrupt, ABEND, or program or Language Environment condition of Severity 1 and above occurs.
  >
  > Debug commands will be entered from the terminal.

  PROMPT
  > The Debug Tool is invoked immediately after Language Environment initialization.

  JBJONES%SESSNA:
  > CODE/370 initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.

- If you want to save the output from your debugging session, issue a command that names a log file. For example,

  ```
  SET LOG ON FILE dbgtool.log;
  ```

  starts logging to a file on the workstation called dbgtool.log. This should be the first command that you enter from the terminal or include in your commands file.

Using CODE/370 in batch mode: To test your stored procedure in batch mode, you must have the CODE/370 MFI Debug Tool installed on the MVS system where the stored procedure runs. To debug your stored procedure in batch mode using the MFI Debug Tool, do the following:

- If you plan to use the Language Environment run-time option TEST to invoke CODE/370, compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.
- Allocate a log data set to receive the output from CODE/370. Put a DD statement for the log data set in the start-up procedure for the stored procedures address space.
- Enter commands in a data set that you want CODE/370 to execute. Put a DD statement for that data set in the start-up procedure for the stored procedures address space. To define the commands data set to CODE/370, specify the commands data set name or DD name in the TEST run-time option. For example,

  ```
  TEST(ALL,TESTDD,PROMPT,*)
  ```

  tells CODE/370 to look for the commands in the data set associated with DD name TESTDD.

  The first command in the commands data set should be:

  ```
  SET LOG ON FILE ddname;
  ```

That command directs output from your debugging session to the log data set you defined in the previous step. For example, if you defined a log data set with DD name INSPLOG in the stored procedures address space start-up procedure, the first command should be:

```
SET LOG ON FILE INSPLOG;
```

- Invoke the Debug Tool. Two possible methods are:
  - Specify the run-time option TEST. The most convenient place to do that is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.
  - Put CEETEST calls in the stored procedure source code. If you use this approach for an existing stored procedure, you must recompile, re-link, and bind it, then issue the STOP PROCEDURE and START PROCEDURE commands to reload the stored procedure.

    You can combine the run-time option TEST with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

For more information on CODE/370, see *CoOperative Development Environment/370: Debug Tool*.

Using the MSGFILE run-time option

Language Environment supports the run-time option MSGFILE, which identifies a JCL DD statement used for writing debugging messages. You can use the MSGFILE option to direct debugging messages to a DASD file or JES spool file. The following considerations apply:

- For each MSGFILE argument, you must add a DD statement to the JCL procedure used to start the DB2 stored procedures address space.
- Execute ALTER PROCEDURE with the RUN OPTIONS parameter to add the MSGFILE option to the list of run-time options for the stored procedure.
- Because multiple TCBs can be active in the DB2 stored procedures address space, you must serialize I/O to the data set associated with the MSGFILE option. For example:
  - To prevent multiple procedures from sharing a data set, each stored procedure can specify a unique DD name with the MSGFILE option.
  - If you debug your applications infrequently or on a DB2 test system, you can serialize I/O by temporarily running the DB2 stored procedures address space with NUMTCB=1 in the stored procedures address space start-up procedure. Ask your system administrator for assistance in doing this.

Using driver applications

You can write a small driver application that calls the stored procedure as a subprogram and passes the parameter list supported by the stored procedure. You can then test and debug the stored procedure as a normal DB2 application under TSO. Now you can use TSO TEST and other commonly used debugging tools.

Using SQL INSERTs

You can use SQL to insert debugging information into a DB2 table. This allows other machines in the network (such as a workstation) to easily access the data in the table using DRDA access.

DB2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Chapter 25. Tuning your queries

This chapter tells you how to improve the performance of your queries. It begins with:
- "General tips and questions"

For more detailed information and suggestions, see:
- "Writing efficient predicates" on page 624
- "Using host variables efficiently" on page 644
- "Writing efficient subqueries" on page 648
- "Using scrollable cursors efficiently" on page 654

If you still have performance problems after you have tried the suggestions in these sections, there are other, more risky techniques you can use. See "Special techniques to influence access path selection" on page 655 for information.

General tips and questions

Recommendation: If you have a query that is performing poorly, first go over the following checklist to see that you have not overlooked some of the basics.

Is the query coded as simply as possible?

Make sure the SQL query is coded as simply and efficiently as possible. Make sure that no unused columns are selected and that there is no unneeded ORDER BY or GROUP BY.

Are all predicates coded correctly?

Indexable predicates: Make sure all the predicates that you think should be indexable are coded so that they can be indexable. Refer to Table 68 on page 629 to see which predicates are indexable and which are not.

Unintentionally redundant or unnecessary predicates: Try to remove any predicates that are unintentionally redundant or not needed; they can slow down performance.

Declared lengths of host variables: Make sure that the declared length of any host variable is no greater than the length attribute of the data column it is compared to. If the declared length is greater, the predicate is stage 2 and cannot be a matching predicate for an index scan.

For example, assume that a host variable and an SQL column are defined as follows:

Assembler declaration	SQL definition
MYHOSTV DS PLn 'value'	COL1 DECIMAL(6,3)

When 'n' is used, the precision of the host variable is '2n-1'. If n = 4 and value = '123.123', then a predicate such as `WHERE COL1 = :MYHOSTV` is not a matching predicate for an index scan because the precisions are different. One way to avoid an inefficient predicate using decimal host variables is to declare the host variable without the 'Ln' option:

```
MYHOSTV  DS P'123.123'
```

This guarantees the same host variable declaration as the SQL column definition.

Are there subqueries in your query?

If your query uses subqueries, see "Writing efficient subqueries" on page 648 to understand how DB2 executes subqueries. There are no absolute rules to follow when deciding how or whether to code a subquery. But these are general guidelines:

- If there are efficient indexes available on the tables in the subquery, then a correlated subquery is likely to be the most efficient kind of subquery.
- If there are no efficient indexes available on the tables in the subquery, then a noncorrelated subquery would likely perform better.
- If there are multiple subqueries in any parent query, make sure that the subqueries are ordered in the most efficient manner.

Consider the following illustration. Assume that there are 1000 rows in MAIN_TABLE.

```
SELECT * FROM MAIN_TABLE
WHERE TYPE IN (subquery 1)
  AND
  PARTS IN (subquery 2);
```

Assuming that subquery 1 and subquery 2 are the same type of subquery (either correlated or noncorrelated), DB2 evaluates the subquery predicates in the order they appear in the WHERE clause. Subquery 1 rejects 10% of the total rows, and subquery 2 rejects 80% of the total rows.

The predicate in subquery 1 (which is referred to as P1) is evaluated 1,000 times, and the predicate in subquery 2 (which is referred to as P2) is evaluated 900 times, for a total of 1,900 predicate checks. However, if the order of the subquery predicates is reversed, P2 is evaluated 1000 times, but P1 is evaluated only 200 times, for a total of 1,200 predicate checks.

It appears that coding P2 before P1 would be more efficient if P1 and P2 take an equal amount of time to execute. However, if P1 is 100 times faster to evaluate than P2, then it might be advisable to code subquery 1 first. If you notice a performance degradation, consider reordering the subqueries and monitoring the results. Consult "Writing efficient subqueries" on page 648 to help you understand what factors make one subquery run more slowly than another.

If you are in doubt, run EXPLAIN on the query with both a correlated and a noncorrelated subquery. By examining the EXPLAIN output and understanding your data distribution and SQL statements, you should be able to determine which form is more efficient.

This general principle can apply to all types of predicates. However, because subquery predicates can potentially be thousands of times more processor- and I/O-intensive than all other predicates, it is most important to make sure they are coded in the correct order.

DB2 always performs all noncorrelated subquery predicates before correlated subquery predicates, regardless of coding order.

Refer to "DB2 predicate manipulation" on page 638 to see in what order DB2 will evaluate predicates and when you can control the evaluation order.

Does your query involve column functions?

If your query involves column functions, make sure that they are coded as simply as possible; this increases the chances that they will be evaluated when the data is retrieved, rather than afterward. In general, a column function performs best when evaluated during data access and next best when evaluated during DB2 sort. Least preferable is to have a column function evaluated after the data has been retrieved. Refer to "When are column functions evaluated? (COLUMN_FN_EVAL)" on page 683 for help in using EXPLAIN to get the information you need.

For column functions to be evaluated during data retrieval, the following conditions must be met for all column functions in the query:
- There must be no sort needed for GROUP BY. Check this in the EXPLAIN output.
- There must be no stage 2 (residual) predicates. Check this in your application.
- There must be no distinct set functions such as COUNT(DISTINCT C1).
- If the query is a join, all set functions must be on the last table joined. Check this by looking at the EXPLAIN output.
- All column functions must be on single columns with no arithmetic expressions.
- The column function is *not* one of the following column functions:
- STDDEV
- STDDEV_SAMP
- VAR
- VAR_SAMP

If your query involves the functions MAX or MIN, refer to "One-fetch access (ACCESSTYPE=I1)" on page 688 to see whether your query could take advantage of that method.

Do you have an input variable in the predicate of a static SQL query?

When host variables or parameter markers are used in a query, the actual values are not known when you bind the package or plan that contains the query. DB2 therefore uses a default filter factor to determine the best access path for an SQL statement. If that access path proves to be inefficient, there are several things you can do to obtain a better access path.

See "Using host variables efficiently" on page 644 for more information.

Do you have a problem with column correlation?

Two columns in a table are said to be correlated if the values in the columns do not vary independently.

DB2 might not determine the best access path when your queries include correlated columns. If you think you have a problem with column correlation, see "Column correlation" on page 641 for ideas on what to do about it.

Can your query be written to use a noncolumn expression?

The following predicate combines a column, SALARY, with values that are not from columns on one side of the operator:

```
WHERE SALARY + (:hv1 * SALARY) > 50000
```

If you rewrite the predicate in the following way, DB2 can evaluate it more efficiently:

```
WHERE SALARY > 50000/(1 + :hv1)
```

In the second form, the column is by itself on one side of the operator, and all the other values are on the other side of the operator. The expression on the right is called a *noncolumn expression*. DB2 can evaluate many predicates with noncolumn expressions at an earlier stage of processing called *stage 1*, so the queries take less time to run.

For more information on noncolumn expressions and stage 1 processing, see "Properties of predicates".

Writing efficient predicates

Definition: Predicates are found in the clauses WHERE, HAVING or ON of SQL statements; they describe attributes of data. They are usually based on the columns of a table and either qualify rows (through an index) or reject rows (returned by a scan) when the table is accessed. The resulting qualified or rejected rows are independent of the access path chosen for that table.

Example: The query below has three predicates: an equal predicate on C1, a BETWEEN predicate on C2, and a LIKE predicate on C3.

```
SELECT * FROM T1
    WHERE C1 = 10 AND
          C2 BETWEEN 10 AND 20 AND
          C3 NOT LIKE 'A%'
```

Effect on access paths: This section explains the effect of predicates on access paths. Because SQL allows you to express the same query in different ways, knowing how predicates affect path selection helps you write queries that access data efficiently.

This section describes:
* "Properties of predicates"
* "General rules about predicate evaluation" on page 627
* "Predicate filter factors" on page 633
* "DB2 predicate manipulation" on page 638
* "Column correlation" on page 641

Properties of predicates

Predicates in a HAVING clause are not used when selecting access paths; hence, in this section the term 'predicate' means a predicate after WHERE or ON.

A predicate influences the selection of an access path because of:
* Its **type**, as described in "Predicate types" on page 625
* Whether it is **indexable**, as described in "Indexable and nonindexable predicates" on page 626
* Whether it is **stage 1** or **stage 2**
* Whether it contains a ROWID column, as described in "Is direct row access possible? (PRIMARY_ACCESSTYPE = D)" on page 677

There are special considerations for "Predicates in the ON clause" on page 627.

Definitions: Predicates are identified as:

Simple or compound

A *compound* predicate is the result of two predicates, whether simple or compound, connected together by AND or OR Boolean operators. All others are *simple*.

Local or join

Local predicates reference only one table. They are local to the table and restrict the number of rows returned for that table. *Join predicates* involve more than one table or correlated reference. They determine the way rows are joined from two or more tables. For examples of their use, see "Interpreting access to two or more tables (join)" on page 689.

Boolean term

Any predicate that is not contained by a compound OR predicate structure is a *Boolean term*. If a Boolean term is evaluated false for a particular row, the whole WHERE clause is evaluated false for that row.

Predicate types

The type of a predicate depends on its operator or syntax, as listed below. The type determines what type of processing and filtering occurs when the predicate is evaluated.

Type   Definition

Subquery

Any predicate that includes another SELECT statement. Example: C1 IN (SELECT C10 FROM TABLE1)

Equal Any predicate that is not a subquery predicate and has an equal operator and no NOT operator. Also included are predicates of the form C1 IS NULL. Example: C1=100

Range

Any predicate that is not a subquery predicate and has an operator in the following list: >, >=, <, <=, LIKE, or BETWEEN. Example: C1>100

IN-list A predicate of the form column IN (list of values). Example: C1 IN (5,10,15)

NOT Any predicate that is not a subquery predicate and contains a NOT operator. Example: COL1 <> 5 or COL1 NOT BETWEEN 10 AND 20.

Example: Influence of type on access paths: The following two examples show how the predicate type can influence DB2's choice of an access path. In each one, assume that a unique index I1 (C1) exists on table T1 (C1, C2), and that all values of C1 are positive integers.

The query,
```
SELECT C1, C2 FROM T1 WHERE C1 >= 0;
```

has a range predicate. However, the predicate does not eliminate any rows of T1. Therefore, it could be determined during bind that a table space scan is more efficient than the index scan.

The query,
```
SELECT * FROM T1 WHERE C1 = 0;
```

has an equal predicate. DB2 chooses the index access in this case, because only one scan is needed to return the result.

Indexable and nonindexable predicates

Definition: *Indexable* predicate types can match index entries; other types cannot. Indexable predicates might not become matching predicates of an index; it depends on the indexes that are available and the access path chosen at bind time.

Examples: If the employee table has an index on the column LASTNAME, the following predicate can be a matching predicate:

```
SELECT * FROM DSN8710.EMP WHERE LASTNAME = 'SMITH';
```

The following predicate cannot be a matching predicate, because it is not indexable.

```
SELECT * FROM DSN8710.EMP WHERE SEX <> 'F';
```

Recommendation: To make your queries as efficient as possible, use indexable predicates in your queries and create suitable indexes on your tables. Indexable predicates allow the possible use of a matching index scan, which is often a very efficient access path.

Stage 1 and stage 2 predicates

Definition: Rows retrieved for a query go through two stages of processing.

1. *Stage 1* predicates (sometimes called *sargable*) can be applied at the first stage.
2. *Stage 2* predicates (sometimes called *nonsargable* or *residual*) cannot be applied until the second stage.

The following items determine whether a predicate is stage 1:

- Predicate syntax

  See Table 68 on page 629 for a list of simple predicates and their types. See Examples of predicate properties for information on compound predicate types.

- Type and length of constants in the predicate

  A simple predicate whose syntax classifies it as stage 1 might not be stage 1 because it contains constants and columns whose types or lengths disagree. For example, the following predicates are not stage 1:
  – CHARCOL='ABCDEFG', where CHARCOL is defined as CHAR(6)
  – SINTCOL>34.5, where SINTCOL is defined as SMALLINT

  The first predicate is not stage 1 because the length of the column is shorter than the length of the constant. The second predicate is not stage 1 because the data types of the column and constant are not the same.

- Whether DB2 evaluates the predicate before or after a join operation. A predicate that is evaluated after a join operation is always a stage 2 predicate.

Examples: All indexable predicates are stage 1. The predicate C1 LIKE %BC is also stage 1, but is not indexable.

Recommendation: Use stage 1 predicates whenever possible.

Boolean term (BT) predicates

Definition: A *Boolean term predicate*, or *BT predicate*, is a simple or compound predicate that, when it is evaluated false for a particular row, makes the entire WHERE clause false for that particular row.

Examples: In the following query P1, P2 and P3 are simple predicates:

```
SELECT * FROM T1 WHERE P1 AND (P2 OR P3);
```
- P1 is a simple BT predicate.

- P2 and P3 are simple non-BT predicates.
- P2 OR P3 is a compound BT predicate.
- P1 AND (P2 OR P3) is a compound BT predicate.

Effect on access paths: In single index processing, only Boolean term predicates are chosen for matching predicates. Hence, only indexable Boolean term predicates are candidates for matching index scans. To match index columns by predicates that are not Boolean terms, DB2 considers multiple index access.

In join operations, Boolean term predicates can reject rows at an earlier stage than can non-Boolean term predicates.

Recommendation: For join operations, choose Boolean term predicates over non-Boolean term predicates whenever possible.

Predicates in the ON clause

The ON clause supplies the join condition in an outer join. For a full outer join, the clause can use only equal predicates. For other outer joins, the clause can use any predicates except predicates that contain subqueries.

For left and right outer joins, and for inner joins, join predicates in the ON clause are treated the same as other stage 1 and stage 2 predicates. A stage 2 predicate in the ON clause is treated as a stage 2 predicate of the inner table.

For full outer join, the ON clause is evaluated during the join operation like a stage 2 predicate.

In an outer join, predicates that are evaluated after the join are stage 2 predicates. Predicates in a table expression can be evaluated before the join and can therefore be stage 1 predicates.

For example, in the following statement,

```
SELECT * FROM (SELECT * FROM DSN8710.EMP
   WHERE EDLEVEL  > 100) AS X FULL JOIN DSN8710.DEPT
      ON X.WORKDEPT  = DSN8710.DEPT.DEPTNO;
```

the predicate "EDLEVEL > 100" is evaluated before the full join and is a stage 1 predicate. For more information on join methods, see "Interpreting access to two or more tables (join)" on page 689.

General rules about predicate evaluation

Recommendations:

1. In terms of resource usage, the earlier a predicate is evaluated, the better.
2. Stage 1 predicates are better than stage 2 predicates because they qualify rows earlier and reduce the amount of processing needed at stage 2.
3. When possible, try to write queries that evaluate the most restrictive predicates first. When predicates with a high filter factor are processed first, unnecessary rows are screened as early as possible, which can reduce processing cost at a later stage. However, a predicate's restrictiveness is only effective among predicates of the same type and the same evaluation stage. For information about filter factors, see "Predicate filter factors" on page 633.

Order of evaluating predicates

Two sets of rules determine the order of predicate evaluation.

The first set:

1. Indexable predicates are applied first. All matching predicates on index key columns are applied first and evaluated when the index is accessed.

   First, stage 1 predicates that have not been picked as matching predicates but still refer to index columns are applied to the index. This is called *index screening*.

2. Other stage 1 predicates are applied next.

   After data page access, stage 1 predicates are applied to the data.

3. Finally, the stage 2 predicates are applied on the returned data rows.

The second set of rules describes the order of predicate evaluation within each of the above stages:

1. All equal predicates a (including column IN *list*, where *list* has only one element).

2. All range predicates and predicates of the form *column* IS NOT NULL

3. All other predicate types are evaluated.

After both sets of rules are applied, predicates are evaluated in the order in which they appear in the query. Because you specify that order, you have some control over the order of evaluation.

Summary of predicate processing

Table 68 on page 629 lists many of the simple predicates and tells whether those predicates are indexable or stage 1. The following terms are used:

- *non subq* means a noncorrelated subquery.
- *cor subq* means a correlated subquery.
- *op* is any of the operators >, >=, <, <=, ¬>, ¬<.
- *value* is a constant, host variable, or special register.
- *pattern* is any character string that does *not* start with the special characters for percent (%) or underscore (_).
- *char* is any character string that does *not* include the special characters for percent (%) or underscore (_).
- *expression* is any expression that contains arithmetic operators, scalar functions, column functions, concatenation operators, columns, constants, host variables, special registers, or date or time expressions.
- *noncol expr* is a noncolumn expression, which is any expression that does not contain a column. That expression can contain arithmetic operators, scalar functions, concatenation operators, constants, host variables, special registers, or date or time expressions.

  An example of a noncolumn expression is

  ```
  CURRENT DATE - 50 DAYS
  ```
- *predicate* is a predicate of any type.

In general, if you form a compound predicate by combining several simple predicates with OR operators, the result of the operation has the same characteristics as the simple predicate that is evaluated latest. For example, if two indexable predicates are combined with an OR operator, the result is indexable. If a

stage 1 predicate and a stage 2 predicate are combined with an OR operator, the result is stage 2.

Table 68. Predicate types and processing

Predicate Type	Index-able?	Stage 1?	Notes
COL = *value*	Y	Y	13
COL = *noncol expr*	Y	Y	9, 11, 12
COL IS NULL	Y	Y	
COL *op value*	Y	Y	
COL *op noncol expr*	Y	Y	9, 11
COL BETWEEN *value1* AND *value2*	Y	Y	
COL BETWEEN *noncol expr1* AND *noncol expr2*	Y	Y	9, 11
value BETWEEN COL1 AND COL2	N	N	
COL BETWEEN COL1 AND COL2	N	N	10
COL BETWEEN *expression1* AND *expression2*	N	N	7
COL LIKE '*pattern*'	Y	Y	6
COL IN (*list*)	Y	Y	14
COL <> *value*	N	Y	8
COL <> *noncol expr*	N	Y	8, 11
COL IS NOT NULL	N	Y	
COL NOT BETWEEN *value1* AND *value2*	N	Y	
COL NOT BETWEEN *noncol expr1* AND *noncol expr2*	N	Y	11
value NOT BETWEEN COL1 AND COL2	N	N	
COL NOT IN (*list*)	N	Y	
COL NOT LIKE ' *char*'	N	Y	6
COL LIKE '%*char*'	N	Y	1, 6
COL LIKE '_*char*'	N	Y	1, 6
COL LIKE *host variable*	Y	Y	2, 6
T1.COL = T2.COL	Y	Y	16
T1.COL *op* T2.COL	Y	Y	3
T1.COL <> T2.COL	N	Y	3
T1.COL1 = T1.COL2	N	N	4
T1.COL1 *op* T1.COL2	N	N	4
T1.COL1 <> T1.COL2	N	N	4
COL=(*non subq*)	Y	Y	15
COL = ANY (*non subq*)	N	N	

Table 68. Predicate types and processing (continued)

Predicate Type	Index-able?	Stage 1?	Notes
COL = ALL (*non subq*)	N	N	
COL *op* (*non subq*)	Y	Y	15
COL *op* ANY (*non subq*)	Y	Y	
COL *op* ALL (*non subq*)	Y	Y	
COL <> (*non subq*)	N	Y	
COL <> ANY (*non subq*)	N	N	
COL <> ALL (*non subq*)	N	N	
COL IN (*non subq*)	Y	Y	
(COL1,...COL*n*) IN (*non subq*)	Y	Y	
COL NOT IN (*non subq*)	N	N	
(COL1,...COL*n*) NOT IN (*non subq*)	N	N	
COL = (*cor subq*)	N	N	5
COL = ANY (*cor subq*)	N	N	
COL = ALL (*cor subq*)	N	N	
COL *op* (*cor subq*)	N	N	5
COL *op* ANY (*cor subq*)	N	N	
COL *op* ALL (*cor subq*)	N	N	
COL <> (*cor subq*)	N	N	5
COL <> ANY (*cor subq*)	N	N	
COL <> ALL (*cor subq*)	N	N	
COL IN (*cor subq*)	N	N	
(COL1,...COL*n*) IN (*cor subq*)	N	N	
COL NOT IN (*cor subq*)	N	N	
(COL1,...COL*n*) NOT IN (*cor subq*)	N	N	
EXISTS (*subq*)	N	N	
NOT EXISTS (*subq*)	N	N	
COL = *expression*	Y	Y	7
expression = *value*	N	N	
expression <> *value*	N	N	
expression op value	N	N	
expression op (subquery)	N	N	

Notes to Table 68 on page 629:

1. Indexable only if an ESCAPE character is specified and used in the LIKE predicate. For example, COL LIKE '+%*char*' ESCAPE '+' is indexable.
2. Indexable only if the pattern in the host variable is an indexable constant (for example, host variable='*char%*').
3. Within each statement, the columns are of the same type. Examples of different column types include:
   - Different data types, such as INTEGER and DECIMAL

- Different numeric column lengths, such as DECIMAL(5,0) and DECIMAL(15,0)
- Different decimal scales, such as DECIMAL(7,3) and DECIMAL(7,4).

The following columns are considered to be of the same types:
- Columns of the same data type but different subtypes.
- Columns of the same data type, but different nullability attributes. (For example, one column accepts nulls but the other does not.)

4. If both COL1 and COL2 are from the same table, access through an index on either one is not considered for these predicates. However, the following query is an exception:

```
SELECT *  FROM T1 A, T1 B WHERE A.C1 = B.C2;
```

By using correlation names, the query treats one table as if it were two separate tables. Therefore, indexes on columns C1 and C2 are considered for access.

5. If the subquery has already been evaluated for a given correlation value, then the subquery might not have to be reevaluated.

6. Not indexable or stage 1 if a field procedure exists on that column.

7. Under any of the following circumstances, the predicate is stage 1 and indexable:
   - COL is of type INTEGER or SMALLINT, and *expression* is of the form:

     ```
     integer-constant1 arithmetic-operator integer-constant2
     ```

   - COL is of type DATE, TIME, or TIMESTAMP, and:
     - *expression* is of any of these forms:

       ```
       datetime-scalar-function(character-constant)
       datetime-scalar-function(character-constant) + labeled-duration
       datetime-scalar-function(character-constant) - labeled-duration
       ```

     - The type of *datetime-scalar-function*(*character-constant*) matches the type of COL.
     - The numeric part of *labeled-duration* is an integer.
     - *character-constant* is:
       - Greater than 7 characters long for the DATE scalar function; for example, '1995-11-30'.
       - Greater than 14 characters long for the TIMESTAMP scalar function; for example, '1995-11-30-08.00.00'.
       - Any length for the TIME scalar function.

8. The processing for WHERE NOT COL = *value* is like that for WHERE COL <> *value*, and so on.

9. If *noncol expr*, *noncol expr1*, or *noncol expr2* is a noncolumn expression of one of these forms, then the predicate is not indexable:
   - *noncol expr* + 0
   - *noncol expr* - 0
   - *noncol expr* * 1
   - *noncol expr* / 1
   - *noncol expr* CONCAT *empty string*

10. COL, COL1, and COL2 can be the same column or different columns. The columns can be in the same table or different tables.

11. To ensure that the predicate is indexable and stage 1, make the data type and length of the column and the data type and length of the result of the noncolumn expression the same. For example, if the predicate is:

COL *op scalar function*

and the scalar function is HEX, SUBSTR, DIGITS, CHAR, or CONCAT, then the type and length of the result of the scalar function and the type and length of the column must be the same for the predicate to be indexable and stage 1.

12. Under these circumstances, the predicate is stage 2:
    * *noncol expr* is a case expression.
    * *non col expr* is the product or the quotient of two noncolumn expressions, that product or quotient is an integer value, and COL is a FLOAT or a DECIMAL column.

13. If COL has the ROWID data type, DB2 tries to use direct row access instead of index access or a table space scan.

14. If COL has the ROWID data type, and an index is defined on COL, DB2 tries to use direct row access instead of index access.

15. Not indexable and not stage 1 if COL is not null and the noncorrelated subquery SELECT clause entry can be null.

16. If the columns are numeric columns, they must have the same data type, length, and precision to be stage 1 and indexable. For character columns, the columns can be of different types and lengths. For example, predicates with the following column types and lengths are stage 1 and indexable:
    * CHAR(5) and CHAR(20)
    * VARCHAR(5) and CHAR(5)
    * VARCHAR(5) and CHAR(20)

Examples of predicate properties

Assume that predicate P1 and P2 are simple, stage 1, indexable predicates:
P1 AND P2 is a compound, stage 1, indexable predicate.
P1 OR P2 is a compound, stage 1 predicate, not indexable except by a union of RID lists from two indexes.

The following examples of predicates illustrate the general rules shown in Table 68 on page 629. In each case, assume that there is an index on columns (C1,C2,C3,C4) of the table and that 0 is the lowest value in each column.

* WHERE C1=5 AND C2=7

  Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

* WHERE C1=5 AND C2>7

  Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

* WHERE C1>5 AND C2=7

  Both predicates are stage 1, but only the first matches the index. A matching index scan could be used with C1 as a matching column.

* WHERE C1=5 OR C2=7

  Both predicates are stage 1 but not Boolean terms. The compound is indexable. When DB2 considers multiple index access for the compound predicate, C1 and C2 can be matching columns. For single index access, C1 and C2 can be only index screening columns.

* WHERE C1=5 OR C2<>7

  The first predicate is indexable and stage 1, and the second predicate is stage 1 but not indexable. The compound predicate is stage 1 and not indexable.

* WHERE C1>5 OR C2=7

Both predicates are stage 1 but not Boolean terms. The compound is indexable. When DB2 considers multiple index access for the compound predicate, C1 and C2 can be matching columns. For single index access, C1 and C2 can be only index screening columns.

- WHERE C1 IN (subquery) AND C2=C1

  Both predicates are stage 2 and not indexable. The index is not considered for matching index access, and both predicates are evaluated at stage 2.

- WHERE C1=5 AND C2=7 AND (C3 + 5) IN (7,8)

  The first two predicates only are stage 1 and indexable. The index is considered for matching index access, and all rows satisfying those two predicates are passed to stage 2 to evaluate the third predicate.

- WHERE C1=5 OR C2=7 OR (C3 + 5) IN (7,8)

  The third predicate is stage 2. The compound predicate is stage 2 and all three predicates are evaluated at stage 2. The simple predicates are not Boolean terms and the compound predicate is not indexable.

- WHERE C1=5 OR (C2=7 AND C3=C4)

  The third predicate is stage 2. The two compound predicates (C2=7 AND C3=C4) and (C1=5 OR (C2=7 AND C3=C4)) are stage 2. All predicates are evaluated at stage 2.

- WHERE (C1>5 OR C2=7) AND C3 = C4

  The compound predicate (C1>5 OR C2=7) is indexable and stage 1. The simple predicate C3=C4 is not stage1; so the index is not considered for matching index access. Rows that satisfy the compound predicate (C1>5 OR C2=7) are passed to stage 2 for evaluation of the predicate C3=C4.

- WHERE T1.COL1=T2.COL1 AND T1.COL2=T2.COL2

  Assume that T1.COL1 and T2.COL1 have the same data types, and T1.COL2 and T2.COL2 have the same data types. If T1.COL1 and T2.COL1 have different nullability attributes, but T1.COL2 and T2.COL2 have the same nullability attributes, and DB2 chooses a merge scan join to evaluate the compound predicate, the compound predicate is stage 1. However, if T1.COL2 and T2.COL2 also have different nullability attributes, and DB2 chooses a merge scan join, the compound predicate is not stage 1.

Predicate filter factors

Definition: The *filter factor* of a predicate is a number between 0 and 1 that estimates the proportion of rows in a table for which the predicate is true. Those rows are said to *qualify* by that predicate.

Example: Suppose that DB2 can determine that column C1 of table T contains only five distinct values: A, D, Q, W and X. In the absence of other information, DB2 estimates that one-fifth of the rows have the value D in column C1. Then the predicate C1='D' has the filter factor 0.2 for table T.

How DB2 uses filter factors: Filter factors affect the choice of access paths by estimating the number of rows qualified by a set of predicates.

For simple predicates, the filter factor is a function of three variables:

1. The literal value in the predicate; for instance, 'D' in the previous example.
2. The operator in the predicate; for instance, '=' in the previous example and '<>' in the negation of the predicate.
3. Statistics on the column in the predicate. In the previous example, those include the information that column T.C1 contains only five values.

Recommendation: You control the first two of those variables when you write a predicate. Your understanding of DB2's use of filter factors should help you write more efficient predicates.

Values of the third variable, statistics on the column, are kept in the DB2 catalog. You can update many of those values, either by running the utility RUNSTATS or by executing UPDATE for a catalog table. For information about using RUNSTATS, . see the discussion of maintaining statistics in the catalog in Part 4 (Volume 1) of *DB2 Administration Guide* For information on updating the catalog manually, see "Updating catalog statistics" on page 663.

If you intend to update the catalog with statistics of your own choice, you should understand how DB2 uses:
- "Default filter factors for simple predicates"
- "Filter factors for uniform distributions"
- "Interpolation formulas" on page 635
- "Filter factors for all distributions" on page 636

Default filter factors for simple predicates
Table 69 lists default filter factors for different types of predicates. DB2 uses those values when no other statistics exist.

Example: The default filter factor for the predicate `C1 = 'D'` is 1/25 (0.04). If D is actually one of only five distinct values in column C1, the default probably does not lead to an optimal access path.

Table 69. DB2 default filter factors by predicate type

Predicate Type	Filter Factor
Col = literal	1/25
Col IS NULL	1/25
Col IN (literal list)	(number of literals)/25
Col *Op* literal	1/3
Col LIKE literal	1/10
Col BETWEEN literal1 and literal2	1/10

Note:
    Op is one of these operators: <, <=, >, >=.
    Literal is any constant value that is known at bind time.

Filter factors for uniform distributions
DB2 uses the filter factors in Table 70 if:

- There is a positive value in column COLCARDF of catalog table SYSIBM.SYSCOLUMNS for the column "Col".
- There are no additional statistics for "Col" in SYSIBM.SYSCOLDIST.

Example: If D is one of only five values in column C1, using RUNSTATS will put the value 5 in column COLCARDF of SYSCOLUMNS. If there are no additional statistics available, the filter factor for the predicate `C1 = 'D'` is 1/5 (0.2).

Table 70. DB2 uniform filter factors by predicate type

Predicate Type	Filter Factor
Col = literal	1/COLCARDF
Col IS NULL	1/COLCARDF

Table 70. DB2 uniform filter factors by predicate type (continued)

Predicate Type	Filter Factor
Col IN (literal list)	number of literals /COLCARDF
Col *Op1* literal	interpolation formula
Col *Op2* literal	interpolation formula
Col LIKE literal	interpolation formula
Col BETWEEN literal1 and literal2	interpolation formula

Note:

> *Op1* is < or <=, and the literal is not a host variable.
> *Op2* is > or >=, and the literal is not a host variable.
> *Literal* is any constant value that is known at bind time.

Filter factors for other predicate types: The examples selected in Table 69 on page 634 and Table 70 on page 634 represent only the most common types of predicates. If P1 is a predicate and F is its filter factor, then the filter factor of the predicate NOT P1 is (1 - F). But, filter factor calculation is dependent on many things, so a specific filter factor cannot be given for all predicate types.

Interpolation formulas

Definition: For a predicate that uses a range of values, DB2 calculates the filter factor by an *interpolation formula*. The formula is based on an estimate of the ratio of the number of values in the range to the number of values in the entire column of the table.

The formulas: The formulas that follow are rough estimates, subject to further modification by DB2. They apply to a predicate of the form `col op. literal`. The value of (Total Entries) in each formula is estimated from the values in columns HIGH2KEY and LOW2KEY in catalog table SYSIBM.SYSCOLUMNS for column *col*: Total Entries = (HIGH2KEY value - LOW2KEY value).

- For the operators < and <=, where the literal is not a host variable:
  (Literal value - LOW2KEY value) / (Total Entries)

- For the operators > and >=, where the literal is not a host variable:
  (HIGH2KEY value - Literal value) / (Total Entries)

- For LIKE or BETWEEN:
  (High literal value - Low literal value) / (Total Entries)

Example: For column C2 in a predicate, suppose that the value of HIGH2KEY is 1400 and the value of LOW2KEY is 200. For C2, DB2 calculates (Total Entries) = 1200.

For the predicate `C1 BETWEEN 800 AND 1100`, DB2 calculates the filter factor F as:
`F = (1100 - 800)/1200 = 1/4 = 0.25`

Interpolation for LIKE: DB2 treats a LIKE predicate as a type of BETWEEN predicate. Two values that bound the range qualified by the predicate are generated from the literal string in the predicate. Only the leading characters found before the escape character ('%' or '_') are used to generate the bounds. So if the escape character is the first character of the string, the filter factor is estimated as 1, and the predicate is estimated to reject no rows.

Defaults for interpolation: DB2 might not interpolate in some cases; instead, it can use a default filter factor. Defaults for interpolation are:

- Relevant only for ranges, including LIKE and BETWEEN predicates
- Used only when interpolation is not adequate
- Based on the value of COLCARDF
- Used whether uniform or additional distribution statistics exist on the column if either of the following conditions is met:
  - The predicate does not contain constants
  - COLCARDF < 4.

Table 71 shows interpolation defaults for the operators <, <=, >, >= and for LIKE and BETWEEN.

Table 71. Default filter factors for interpolation

COLCARDF	Factor for Op	Factor for LIKE or BETWEEN
≥100,000,000	1/10,000	3/100,000
≥10,000,000	1/3,000	1/10,000
≥1,000,000	1/1,000	3/10,000
≥100,000	1/300	1/1,000
≥10,000	1/100	3/1,000
≥1,000	1/30	1/100
≥100	1/10	3/100
≥0	1/3	1/10

Note: Op is one of these operators: <, <=, >, >=.

Filter factors for all distributions

RUNSTATS can generate additional statistics for a column or set of concatenated key columns of an index. DB2 can use that information to calculate filter factors. DB2 collects two kinds of distribution statistics:

Frequency
> The percentage of rows in the table that contain a value for a column or combination of values for concatenated columns

Cardinality
> The number of distinct values in concatenated columns

When they are used: Table 72 lists the types of predicates on which these statistics are used.

Table 72. Predicates for which distribution statistics are used

Type of statistic	Single column or concatenated columns	Predicates
Frequency	Single	COL=*literal* COL IS NULL COL IN (*literal-list*) COL *op literal* COL BETWEEN *literal* AND *literal*
Frequency	Concatenated	COL=*literal*

Table 72. Predicates for which distribution statistics are used (continued)

Type of statistic	Single column or concatenated columns	Predicates
Cardinality	Single	COL=*literal* COL IS NULL COL IN (*literal-list*) COL *op literal* COL BETWEEN *literal* AND *literal* COL=*host-variable* COL1=COL2
Cardinality	Concatenated	COL=*literal* COL=:*host-variable* COL1=COL2

Note: *op* is one of these operators: <, <=, >, >=.

How they are used: Columns COLVALUE and FREQUENCYF in table
SYSCOLDIST contain distribution statistics. Regardless of the number of values in
those columns, running RUNSTATS deletes the existing values and inserts rows for
the most frequent values. If you run RUNSTATS without the FREQVAL option,
RUNSTATS inserts rows for the 10 most frequent values for the first column of the
specified index. If you run RUNSTATS with the FREQVAL option and its two
keywords, NUMCOLS and COUNT, RUNSTATS inserts rows for concatenated
columns of an index. NUMCOLS specifies the number of concatenated index
columns. COUNT specifies the number of most frequent values. See Part 2 of *DB2
Utility Guide and Reference* for more information about RUNSTATS. DB2 uses the
frequencies in column FREQUENCYF for predicates that use the values in column
COLVALUE and assumes that the remaining data are uniformly distributed.

Example: Filter factor for a single column

Suppose that the predicate is C1 IN ('3','5') and that SYSCOLDIST contains
these values for column C1:

```
COLVALUE    FREQUENCYF
  '3'          .0153
  '5'          .0859
  '8'          .0627
```

The filter factor is .0153 + .0859 = .1012.

Example: Filter factor for correlated columns

Suppose that columns C1 and C2 are correlated and are concatenated columns of
an index. Suppose also that the predicate is C1='3' AND C2='5' and that
SYSCOLDIST contains these values for columns C1 and C2:

```
COLVALUE    FREQUENCYF
 '1' '1'       .1176
 '2' '2'       .0588
 '3' '3'       .0588
 '3' '5'       .1176
 '4' '4'       .0588
 '5' '3'       .1764
 '5' '5'       .3529
 '6' '6'       .0588
```

The filter factor is .1176.

DB2 predicate manipulation

In some specific cases, DB2 either modifies some predicates, or generates extra predicates. Although these modifications are transparent to you, they have a direct impact on the access path selection and your PLAN_TABLE results. This is because DB2 always uses an index access path when it is cost effective. Generating extra predicates provides more indexable predicates potentially, which creates more chances for an efficient index access path.

Therefore, to understand your PLAN_TABLE results, you must understand how DB2 manipulates predicates. The information in Table 68 on page 629 is also helpful.

Predicate modifications for IN-list predicates

If an IN-list predicate has only one item in its list, the predicate becomes an EQUAL predicate.

A set of simple, Boolean term, equal predicates on the same column that are connected by OR predicates can be converted into an IN-list predicate. For example: `C1=5 or C1=10 or C1=15` converts to `C1 IN (5,10,15)`.

When DB2 simplifies join operations

Because full outer joins are less efficient than left or right joins, and left and right joins are less efficient than inner joins, you should always try to use the simplest type of join operation in your queries. However, if DB2 encounters a join operation that it can simplify, it attempts to do so. In general, DB2 can simplify a join operation when the query contains a predicate or an ON clause that eliminates the null values that are generated by the join operation.

For example, consider this query:

```
SELECT * FROM T1 X FULL JOIN T2 Y
  ON X.C1=Y.C1
  WHERE X.C2 > 12;
```

The outer join operation gives you these result table rows:

- The rows with matching values of C1 in tables T1 and T2 (the inner join result)
- The rows from T1 where C1 has no corresponding value in T2
- The rows from T2 where C1 has no corresponding value in T1

However, when you apply the predicate, you remove all rows in the result table that came from T2 where C1 has no corresponding value in T1. DB2 transforms the full join into a left join, which is more efficient:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  ON X.C1=Y.C1
  WHERE X.C2 > 12;
```

In the following example, the predicate, X.C2>12, filters out all null values that result from the right join:

```
SELECT * FROM T1 X RIGHT JOIN T2 Y
  ON X.C1=Y.C1
  WHERE X.C2>12;
```

Therefore, DB2 can transform the right join into a more efficient inner join without changing the result:

```
SELECT * FROM T1 X INNER JOIN T2 Y
  ON X.C1=Y.C1
  WHERE X.C2>12;
```

The predicate that follows a join operation must have the following characteristics before DB2 transforms an outer join into a simpler outer join or into an inner join:
- The predicate is a Boolean term predicate.
- The predicate is false if one table in the join operation supplies a null value for all of its columns.

These predicates are examples of predicates that can cause DB2 to simplify join operations:
- T1.C1 > 10
- T1.C1 IS NOT NULL
- T1.C1 > 10 OR T1.C2 > 15
- T1.C1 > T2.C1
- T1.C1 IN (1,2,4)
- T1.C1 LIKE 'ABC%'
- T1.C1 BETWEEN 10 AND 100
- 12 BETWEEN T1.C1 AND 100

The following example shows how DB2 can simplify a join operation because the query contains an ON clause that eliminates rows with unmatched values:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  FULL JOIN T3 Z ON Y.C1=Z.C1
  ON X.C1=Y.C1;
```

Because the last ON clause eliminates any rows from the result table for which column values that come from T1 or T2 are null, DB2 can replace the full join with a more efficient left join to achieve the same result:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  LEFT JOIN T3 Z ON Y.C1=Z.C1
  ON X.C1=Y.C1;
```

There is one case in which DB2 transforms a full outer join into a left join when you cannot write code to do it. This is the case where a view specifies a full outer join, but a subsequent query on that view requires only a left outer join. For example, consider this view:

```
CREATE VIEW V1 (C1,T1C2,T2C2) AS
  SELECT COALESCE(T1.C1, T2.C1), T1.C2, T2.C2
  FROM T1 X FULL JOIN T2 Y
  ON T1.C1=T2.C1;
```

This view contains rows for which values of C2 that come from T1 are null. However, if you execute the following query, you eliminate the rows with null values for C2 that come from T1:

```
SELECT * FROM V1
  WHERE T1C2 > 10;
```

Therefore, for this query, a left join between T1 and T2 would have been adequate. DB2 can execute this query as if the view V1 was generated with a left outer join so that the query runs more efficiently.

Predicates generated through transitive closure
When the set of predicates that belong to a query logically imply other predicates, DB2 can generate additional predicates to provide more information for access path selection.

Rules for generating predicates: For single-table or inner join queries, DB2 generates predicates for transitive closure if:

- The query has an equal type predicate: COL1=COL2. This could be:
  - A local predicate
  - A join predicate
- The query also has a Boolean term predicate on one of the columns in the first predicate with one of the following formats:
  - COL1 *op value*

    op is =, <>, >, >=, <, or <=.

    value is a constant, host variable, or special register.
  - COL1 (NOT) BETWEEN *value1* AND *value2*
  - COL1=COL3

For outer join queries, DB2 generates predicates for transitive closure if the query has an ON clause of the form COL1=COL2 and a before join predicate that has one of the following formats:
- COL1 *op value*

  op is =, <>, >, >=, <, or <=
- COL1 (NOT) BETWEEN *value1* AND *value2*

DB2 generates a transitive closure predicate for an outer join query only if the generated predicate does not reference the table with unmatched rows. That is, the generated predicate cannot reference the left table for a left outer join or the right table for a right outer join.

When a predicate meets the the transitive closure conditions, DB2 generates a new predicate, whether or not it already exists in the WHERE clause.

The generated predicates have one of the following formats:
- COL *op value*

  op is =, <>, >, >=, <, or <=.

  value is a constant, host variable, or special register.
- COL (NOT) BETWEEN *value1* AND *value2*
- COL1=COL2 (for single-table or inner join queries only)

Example of transitive closure for an inner join: Suppose that you have written this query, which meets the conditions for transitive closure:

```
SELECT * FROM T1, T2
  WHERE T1.C1=T2.C1 AND
  T1.C1>10;
```

DB2 generates an additional predicate to produce this query, which is more efficient:

```
SELECT * FROM T1, T2
  WHERE T1.C1=T2.C1 AND
  T1.C1>10 AND
  T2.C1>10;
```

Example of transitive closure for an outer join: Suppose that you have written this outer join query:

```
SELECT * FROM (SELECT * FROM T1 WHERE T1.C1>10) X
  LEFT JOIN T2
  ON X.C1 = T2.C1;
```

The before join predicate, T1.C1>10, meets the conditions for transitive closure, so DB2 generates this query:

```
SELECT * FROM
  (SELECT * FROM T1 WHERE T1.C1>10 AND T2.C1>10) X
  LEFT JOIN T2
  ON X.C1 = T2.C1;
```

Predicate redundancy: A predicate is redundant if evaluation of other predicates in the query already determines the result that the predicate provides. You can specify redundant predicates or DB2 can generate them. DB2 does not determine that any of your query predicates are redundant. All predicates that you code are evaluated at execution time regardless of whether they are redundant. If DB2 generates a redundant predicate to help select access paths, that predicate is ignored at execution.

Adding extra predicates: DB2 performs predicate transitive closure only on equal and range predicates. Other types of predicates, such as IN or LIKE predicates, might be needed in the following case:

```
SELECT * FROM T1,T2
   WHERE T1.C1=T2.C1
     AND T1.C1 LIKE 'A%';
```

In this case, add the predicate T2.C1 LIKE 'A%'.

Column correlation

Two columns of data, A and B of a single table, are correlated if the values in column A do not vary independently of the values in column B.

The following is an excerpt from a large single table. Columns CITY and STATE are highly correlated, and columns DEPTNO and SEX are entirely independent.

TABLE CREWINFO

CITY	STATE	DEPTNO	SEX	EMPNO	ZIPCODE
Fresno	CA	A345	F	27375	93650
Fresno	CA	J123	M	12345	93710
Fresno	CA	J123	F	93875	93650
Fresno	CA	J123	F	52325	93792
New York	NY	J123	M	19823	09001
New York	NY	A345	M	15522	09530
Miami	FL	B499	M	83825	33116
Miami	FL	A345	F	35785	34099
Los Angeles	CA	X987	M	12131	90077
Los Angeles	CA	A345	M	38251	90091

In this simple example, for every value of column CITY that equals 'FRESNO', there is the same value in column STATE ('CA').

How to detect column correlation

The first indication that column correlation is a problem is because of poor response times when DB2 has chosen an inappropriate access path. If you suspect two columns in a table (CITY and STATE in table CREWINFO) are correlated, then you can issue the following SQL queries that reflect the relationships between the columns:

```
SELECT COUNT (DISTINCT CITY) FROM CREWINFO;  (RESULT1)
SELECT COUNT (DISTINCT STATE) FROM CREWINFO;   (RESULT2)
```

The result of the count of each distinct column is the value of COLCARDF in the DB2 catalog table SYSCOLUMNS. Multiply the above two values together to get a preliminary result:

```
RESULT1 x RESULT2   =    ANSWER1
```

Then issue the following SQL statement:

```
SELECT COUNT(*) FROM
  (SELECT DISTINCT CITY,STATE
   FROM CREWINFO) AS V1;          (ANSWER2)
```

Compare the result of the above count (ANSWER2) with ANSWER1. If ANSWER2 is less than ANSWER1, then the suspected columns are correlated.

Impacts of column correlation

DB2 might not determine the best access path, table order, or join method when your query uses columns that are highly correlated. Column correlation can make the estimated cost of operations cheaper than they actually are. Column correlation affects both single table queries and join queries.

Column correlation on the best matching columns of an index: The following query selects rows with females in department A345 from Fresno, California. There are 2 indexes defined on the table, Index 1 (CITY,STATE,ZIPCODE) and Index 2 (DEPTNO,SEX).

Query 1

```
SELECT ... FROM CREWINFO WHERE
  CITY = 'FRESNO' AND STATE = 'CA'         (PREDICATE1)
  AND DEPTNO = 'A345' AND SEX = 'F';       (PREDICATE2)
```

Consider the two compound predicates (labeled PREDICATE1 and PREDICATE2), their actual filtering effects (the proportion of rows they select), and their DB2 filter factors. Unless the proper catalog statistics are gathered, the filter factors are calculated as if the columns of the predicate are entirely independent (not correlated).

Table 73. Effects of column correlation on matching columns

	INDEX 1	INDEX 2
Matching Predicates	Predicate1 CITY=FRESNO AND STATE=CA	Predicate2 DEPTNO=A345 AND SEX=F
Matching Columns	2	2
DB2 estimate for matching columns (Filter Factor)	column=CITY, COLCARDF=4 Filter Factor=1/4 column=STATE, COLCARDF=3 Filter Factor=1/3	column=DEPTNO, COLCARDF=4 Filter Factor=1/4 column=SEX, COLCARDF=2 Filter Factor=1/2
Compound Filter Factor for matching columns	1/4 × 1/3 = 0.083	1/4 × 1/2 = 0.125
Qualified leaf pages based on DB2 estimations	0.083 × 10 = 0.83 INDEX CHOSEN (.8 < 1.25)	0.125 × 10 = 1.25
Actual filter factor based on data distribution	4/10	2/10
Actual number of qualified leaf pages based on compound predicate	4/10 × 10 = 4	2/10 × 10 = 2 BETTER INDEX CHOICE (2 < 4)

DB2 chooses an index that returns the fewest rows, partly determined by the smallest filter factor of the matching columns. Assume that filter factor is the only influence on the access path. The combined filtering of columns CITY and STATE seems very good, whereas the matching columns for the second index do not seem to filter as much. Based on those calculations, DB2 chooses Index 1 as an access path for Query 1.

The problem is that the filtering of columns CITY and STATE should not look good. Column STATE does almost no filtering. Since columns DEPTNO and SEX do a better job of filtering out rows, DB2 should favor Index 2 over Index 1.

Column correlation on index screening columns of an index: Correlation might also occur on nonmatching index columns, used for index screening. See "Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)" on page 686 for more information. Index screening predicates help reduce the number of data rows that qualify while scanning the index. However, if the index screening predicates are correlated, they do not filter as many data rows as their filter factors suggest. To illustrate this, use the same Query 1 (see page 642) with the following indexes on table CREWINFO (page 641):

```
Index 3 (EMPNO,CITY,STATE)
Index 4 (EMPNO,DEPTNO,SEX)
```

In the case of Index 3, because the columns CITY and STATE of Predicate 1 are correlated, the index access is not improved as much as estimated by the screening predicates and therefore Index 4 might be a better choice. (Note that index screening also occurs for indexes with matching columns greater than zero.)

Multiple table joins: In Query 2, an additional table is added to the original query (see Query 1 on page 642) to show the impact of column correlation on join queries.

TABLE DEPTINFO

```
CITY           STATE   MANAGER   DEPT    DEPTNAME
------------------------------------------------------
FRESNO         CA      SMITH     J123    ADMIN
LOS ANGELES    CA      JONES     A345    LEGAL
```

Query 2
```
SELECT ... FROM CREWINFO T1,DEPTINFO T2
   WHERE T1.CITY = 'FRESNO' AND T1.STATE='CA'          (PREDICATE 1)
   AND T1.DEPTNO = T2.DEPT AND T2.DEPTNAME = 'LEGAL';
```

The order that tables are accessed in a join statement affects performance. The estimated combined filtering of Predicate1 is lower than its actual filtering. So table CREWINFO might look better as the first table accessed than it should.

Also, due to the smaller estimated size for table CREWINFO, a nested loop join might be chosen for the join method. But, if many rows are selected from table CREWINFO because Predicate1 does not filter as many rows as estimated, then another join method might be better.

What to do about column correlation
If column correlation is causing DB2 to choose an inappropriate access path, try one of these techniques to alter the access path:
- If the correlated columns are concatenated key columns of an index, run the utility RUNSTATS with options KEYCARD and FREQVAL. This is the preferred technique.

- Update the catalog statistics manually.
- Use SQL that forces access through a particular index.

The last two techniques are discussed in "Special techniques to influence access path selection" on page 655.

The utility RUNSTATS collects the statistics DB2 needs to make proper choices about queries. With RUNSTATS, you can collect statistics on the concatenated key columns of an index and the number of distinct values for those concatenated columns. This gives DB2 accurate information to calculate the filter factor for the query.

For example, RUNSTATS collects statistics that benefit queries like this:

```
SELECT * FROM T1
 WHERE C1 = 'a' AND C2 = 'b' AND C3 = 'c' ;
```

where:
- The first three index keys are used (MATCHCOLS = 3).
- An index exists on C1, C2, C3, C4, C5.
- Some or all of the columns in the index are correlated in some way.

See Part 5 (Volume 2) of *DB2 Administration Guide* for information on using RUNSTATS to influence access path selection.

Using host variables efficiently

Host variables require default filter factors: When you bind a static SQL statement that contains host variables, DB2 uses a default filter factor to determine the best access path for the SQL statement. For more information on filter factors, including default values, see "Predicate filter factors" on page 633.

DB2 often chooses an access path that performs well for a query with several host variables. However, in a new release or after maintenance has been applied, DB2 might choose a new access path that does not perform as well as the old access path. In most cases, the change in access paths is due to the default filter factors, which might lead DB2 to optimize the query in a different way.

There are two ways to change the access path for a query that contains host variables:
- Bind the package or plan that contains the query with the option REOPT(VARS).
- Rewrite the query.

Using REOPT(VARS) to change the access path at run time

Specify the bind option REOPT(VARS) when you want DB2 to determine access paths at both bind time and run time for statements that contain one or more of the following:
- host variables
- parameter markers
- special registers

At run time, DB2 uses the values in those variables to determine the access paths.

Because there is a performance cost to reoptimizing the access path at run time, you should use the bind option REOPT(VARS) only on packages or plans containing statements that perform poorly.

Be careful when using REOPT(VARS) for a statement executed in a loop; the reoptimization occurs with every execution of that statement. However, if you are using a cursor, you can put the FETCH statements in a loop because the reoptimization only occurs when the cursor is opened.

To use REOPT(VARS) most efficiently, first determine which SQL statements in your applications perform poorly. Separate the code containing those statements into units that you bind into packages with the option REOPT(VARS). Bind the rest of the code into packages using NOREOPT(VARS). Then bind the plan with the option NOREOPT(VARS). Only statements in the packages bound with REOPT(VARS) are candidates for reoptimization at run time.

To determine which queries in plans and packages bound with REOPT(VARS) will be reoptimized at run time, execute the following SELECT statements:

```
SELECT PLNAME,
  CASE WHEN STMTNOI <> 0
   THEN STMTNOI
   ELSE STMTNO
  END AS STMTNUM,
  SEQNO, TEXT
  FROM   SYSIBM.SYSSTMT
  WHERE STATUS IN ('B','F','G','J')
  ORDER BY PLNAME, STMTNUM, SEQNO;

SELECT COLLID, NAME, VERSION,
  CASE WHEN STMTNOI <> 0
   THEN STMTNOI
   ELSE STMTNO
  END AS STMTNUM,
  SEQNO, STMT
  FROM SYSIBM.SYSPACKSTMT
  WHERE STATUS IN ('B','F','G','J')
  ORDER BY COLLID, NAME, VERSION, STMTNUM, SEQNO;
```

If you specify the bind option VALIDATE(RUN), and a statement in the plan or package is not bound successfully, that statement is incrementally bound at run time. If you also specify the bind option REOPT(VARS), DB2 reoptimizes the access path during the incremental bind.

To determine which plans and packages have statements that will be incrementally bound, execute the following SELECT statements:

```
SELECT DISTINCT NAME
  FROM SYSIBM.SYSSTMT
  WHERE STATUS = 'F' OR STATUS = 'H';

SELECT DISTINCT COLLID, NAME, VERSION
  FROM   SYSIBM.SYSPACKSTMT
  WHERE STATUS = 'F' OR STATUS = 'H';
```

Rewriting queries to influence access path selection

The examples that follow identify potential performance problems and offer suggestions for tuning the queries. However, before you rewrite any query, you should consider whether the bind option REOPT(VARS) can solve your access path problems. See "Using REOPT(VARS) to change the access path at run time" on page 644 for more information on REOPT(VARS).

Example 1: An equal predicate

An equal predicate has a default filter factor of 1/COLCARDF. The actual filter factor might be quite different.

Query:
```
SELECT * FROM DSN8710.EMP
 WHERE SEX = :HV1;
```

Assumptions: Because there are only two different values in column SEX, 'M' and 'F', the value COLCARDF for SEX is 2. If the numbers of male and female employees are not equal, the actual filter factor of 1/2 is larger or smaller than the default, depending on whether :HV1 is set to 'M' or 'F'.

Recommendation: One of these two actions can improve the access path:

• Bind the package or plan that contains the query with the option REOPT(VARS). This action causes DB2 to reoptimize the query at run time, using the input values you provide.

• Write predicates to influence DB2's selection of an access path, based on your knowledge of actual filter factors. For example, you can break the query above into three different queries, two of which use constants. DB2 can then determine the exact filter factor for most cases when it binds the plan.

```
SELECT (HV1);
  WHEN ('M')
    DO;
      EXEC SQL SELECT * FROM DSN8710.EMP
       WHERE SEX = 'M';
    END;
  WHEN ('F')
    DO;
      EXEC SQL SELECT * FROM DSN8710.EMP
       WHERE SEX = 'F';
    END;
  OTHERWISE
    DO:
      EXEC SQL SELECT * FROM DSN8710.EMP
       WHERE SEX = :HV1;
    END;
END;
```

Example 2: Known ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:
```
SELECT * FROM T1
      WHERE C1 BETWEEN :HV1 AND :HV2
        AND C2 BETWEEN :HV3 AND :HV4;
```

Assumptions: You know that:

• The application always provides a narrow range on C1 and a wide range on C2.

• The desired access path is through index T1X1.

Recommendation: If DB2 does not choose T1X1, rewrite the query as follows, so that DB2 does not choose index T1X2 on C2:
```
SELECT * FROM T1
      WHERE C1 BETWEEN :HV1 AND :HV2
        AND (C2 BETWEEN :HV3 AND :HV4 OR 0=1);
```

Example 3: Variable ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
         AND C2 BETWEEN :HV3 AND :HV4;
```

Assumptions: You know that the application provides both narrow and wide ranges on C1 and C2. Hence, default filter factors do not allow DB2 to choose the best access path in all cases. For example, a small range on C1 favors index T1X1 on C1, a small range on C2 favors index T1X2 on C2, and wide ranges on both C1 and C2 favor a table space scan.

Recommendation: If DB2 does not choose the best access path, try either of the following changes to your application:

- Use a dynamic SQL statement and embed the ranges of C1 and C2 in the statement. With access to the actual range values, DB2 can estimate the actual filter factors for the query. Preparing the statement each time it is executed requires an extra step, but it can be worthwhile if the query accesses a large amount of data.
- Include some simple logic to check the ranges of C1 and C2, and then execute one of these static SQL statements, based on the ranges of C1 and C2:

```
SELECT * FROM T1 WHERE C1 BETWEEN :HV1 AND :HV2
                  AND  (C2 BETWEEN :HV3 AND :HV4 OR 0=1);

SELECT * FROM T1 WHERE C2 BETWEEN :HV3 AND :HV4
                  AND  (C1 BETWEEN :HV1 AND :HV2 OR 0=1);

SELECT * FROM T1 WHERE (C1 BETWEEN :HV1 AND :HV2 OR 0=1)
                  AND  (C2 BETWEEN :HV3 AND :HV4 OR 0=1);
```

Example 4: ORDER BY

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
       ORDER BY C2;
```

In this example, DB2 could choose one of the following actions:

- Scan index T1X1 and then sort the results by column C2
- Scan the table space in which T1 resides and then sort the results by column C2
- Scan index T1X2 and then apply the predicate to each row of data, thereby avoiding the sort

Which choice is best depends on the following factors:
- The number of rows that satisfy the range predicate
- Which index has the higher cluster ratio

If the actual number of rows that satisfy the range predicate is significantly different from the estimate, DB2 might not choose the best access path.

Assumptions: You disagree with DB2's choice.

Recommendation: In your application, use a dynamic SQL statement and embed the range of C1 in the statement. That allows DB2 to use the actual filter factor rather than the default, but requires extra processing for the PREPARE statement.

Example 5: A join operation

Tables A, B, and C each have indexes on columns C1, C2, C3, and C4.

Query:

```
SELECT * FROM A, B, C
        WHERE A.C1 = B.C1
          AND A.C2 = C.C2
          AND A.C2 BETWEEN :HV1 AND :HV2
          AND A.C3 BETWEEN :HV3 AND :HV4
          AND A.C4 < :HV5
          AND B.C2 BETWEEN :HV6 AND :HV7
          AND B.C3 < :HV8
          AND C.C2 < :HV9;
```

Assumptions: The actual filter factors on table A are much larger than the default factors. Hence, DB2 underestimates the number of rows selected from table A and wrongly chooses that as the first table in the join.

Recommendations: You can:

- Reduce the estimated size of Table A by adding predicates
- Disfavor any index on the join column by making the join predicate on table A nonindexable

The query below illustrates the second of those choices.

```
SELECT * FROM T1 A, T1 B, T1 C
        WHERE (A.C1 = B.C1 OR 0=1)
          AND A.C2 = C.C2
          AND A.C2 BETWEEN :HV1 AND :HV2
          AND A.C3 BETWEEN :HV3 AND :HV4
          AND A.C4 < :HV5
          AND B.C2 BETWEEN :HV6 AND :HV7
          AND B.C3 < :HV8
          AND C.C2 < :HV9;
```

The result of making the join predicate between A and B a nonindexable predicate (which cannot be used in single index access) disfavors the use of the index on column C1. This, in turn, might lead DB2 to access table A or B first. Or, it might lead DB2 to change the access type of table A or B, thereby influencing the join sequence of the other tables.

Writing efficient subqueries

Definitions: A *subquery* is a SELECT statement within the WHERE or HAVING clause of another SQL statement.

Decision needed: You can often write two or more SQL statements that achieve identical results, particularly if you use subqueries. The statements have different access paths, however, and probably perform differently.

Topic overview: The topics that follow describe different methods to achieve the results intended by a subquery and tell what DB2 does for each method. The information should help you estimate what method performs best for your query.

The first two methods use different types of subqueries:
- "Correlated subqueries" on page 649
- "Noncorrelated subqueries" on page 650

A subquery can sometimes be transformed into a join operation. Sometimes DB2 does that to improve the access path, and sometimes you can get better results by doing it yourself. The third method is:

- "Subquery transformation into join" on page 651

  Finally, for a comparison of the three methods as applied to a single task, see:

- "Subquery tuning" on page 653

Correlated subqueries

Definition: A *correlated* subquery refers to at least one column of the outer query.

Any predicate that contains a correlated subquery is a stage 2 predicate.

Example: In the following query, the correlation name, X, illustrates the subquery's reference to the outer query block.

```
SELECT * FROM DSN8710.EMP X
  WHERE  JOB = 'DESIGNER'
    AND  EXISTS (SELECT 1
                 FROM   DSN8710.PROJ
                 WHERE  DEPTNO = X.WORKDEPT
                    AND  MAJPROJ = 'MA2100');
```

What DB2 does: A correlated subquery is evaluated for each qualified row of the outer query that is referred to. In executing the example, DB2:

1. Reads a row from table EMP where JOB='DESIGNER'.
2. Searches for the value of WORKDEPT from that row, in a table stored in memory.

   The in-memory table saves executions of the subquery. If the subquery has already been executed with the value of WORKDEPT, the result of the subquery is in the table and DB2 does not execute it again for the current row. Instead, DB2 can skip to step 5.
3. Executes the subquery, if the value of WORKDEPT is not in memory. That requires searching the PROJ table to check whether there is any project, where MAJPROJ is 'MA2100', for which the current WORKDEPT is responsible.
4. Stores the value of WORKDEPT and the result of the subquery in memory.
5. Returns the values of the current row of EMP to the application.

DB2 repeats this whole process for each qualified row of the EMP table.

Notes on the in-memory table: The in-memory table is applicable if the operator of the predicate that contains the subquery is one of the following operators:

<, <=, >, >=, =, <>, EXISTS, NOT EXISTS

The table is not used, however, if:

- There are more than 16 correlated columns in the subquery
- The sum of the lengths of the correlated columns is more than 256 bytes
- There is a unique index on a subset of the correlated columns of a table from the outer query

The in-memory table is a wrap-around table and does not guarantee saving the results of all possible duplicated executions of the subquery.

Noncorrelated subqueries

Definition: A *noncorrelated* subquery makes no reference to outer queries.

Example:
```
SELECT * FROM DSN8710.EMP
  WHERE  JOB = 'DESIGNER'
    AND  WORKDEPT IN (SELECT DEPTNO
                        FROM   DSN8710.PROJ
                        WHERE  MAJPROJ = 'MA2100');
```

What DB2 does: A noncorrelated subquery is executed once when the cursor is opened for the query. What DB2 does to process it depends on whether it returns a single value or more than one value. The query in the example above can return more than one value.

Single-value subqueries
When the subquery is contained in a predicate with a simple operator, the subquery is required to return 1 or 0 rows. The simple operator can be one of the following operators:

<, <=, >, >=, =, <>, EXISTS, NOT EXISTS

The following noncorrelated subquery returns a single value:
```
  SELECT *
  FROM   DSN8710.EMP
  WHERE  JOB = 'DESIGNER'
    AND  WORKDEPT <= (SELECT MAX(DEPTNO)
                        FROM   DSN8710.PROJ);
```

What DB2 does: When the cursor is opened, the subquery executes. If it returns more than one row, DB2 issues an error. The predicate that contains the subquery is treated like a simple predicate with a constant specified, for example, WORKDEPT <= *'value'*.

Stage 1 and stage 2 processing: The rules for determining whether a predicate with a noncorrelated subquery that returns a single value is stage 1 or stage 2 are generally the same as for the same predicate with a single variable. However, the predicate is stage 2 if:
- The value returned by the subquery is nullable and the column of the outer query is not nullable.
- The data type of the subquery is higher than that of the column of the outer query. For example, the following predicate is stage 2:
  ```
  WHERE SMALLINT_COL < (SELECT INTEGER_COL FROM ...
  ```

Multiple-value subqueries
A subquery can return more than one value if the operator is one of the following:
    op ANY *op* ALL *op* SOME IN EXISTS

where *op* is any of the operators >, >=, <, or <=.

What DB2 does: If possible, DB2 reduces a subquery that returns more than one row to one that returns only a single row. That occurs when there is a range comparison along with ANY, ALL, or SOME. The following query is an example:

```
SELECT * FROM DSN8710.EMP
   WHERE  JOB = 'DESIGNER'
     AND  WORKDEPT <= ANY (SELECT DEPTNO
                           FROM   DSN8710.PROJ
                           WHERE  MAJPROJ = 'MA2100');
```

DB2 calculates the maximum value for DEPTNO from table DSN8710.PROJ and removes the ANY keyword from the query. After this transformation, the subquery is treated like a single-value subquery.

That transformation can be made with a *maximum value* if the range operator is:
- > or >= with the quantifier ALL
- < or <= with the quantifier ANY or SOME

The transformation can be made with a *minimum value* if the range operator is:
- < or <= with the quantifier ALL
- > or >= with the quantifier ANY or SOME

The resulting predicate is determined to be stage 1 or stage 2 by the same rules as for the same predicate with a single-valued subquery.

When a subquery is sorted: A noncorrelated subquery is sorted in descending order when the comparison operator is IN, NOT IN, = ANY, <> ANY, = ALL, or <> ALL. The sort enhances the predicate evaluation, reducing the amount of scanning on the subquery result. When the value of the subquery becomes smaller or equal to the expression on the left side, the scanning can be stopped and the predicate can be determined to be true or false.

When the subquery result is a character data type and the left side of the predicate is a datetime data type, then the result is placed in a work file without sorting. For some noncorrelated subqueries using the above comparison operators, DB2 can more accurately pinpoint an entry point into the work file, thus further reducing the amount of scanning that is done.

Results from EXPLAIN: For information about the result in a plan table for a subquery that is sorted, see "When are column functions evaluated? (COLUMN_FN_EVAL)" on page 683.

Subquery transformation into join

For a SELECT, UPDATE, or DELETE statement, DB2 can sometimes transform a subquery into a join between the result table of a subquery and the result table of an outer query.

For a SELECT statement, DB2 does the transformation if the following conditions are true:
- The transformation does not introduce redundancy.
- The subquery appears in a WHERE clause.
- The subquery does not contain GROUP BY, HAVING, or column functions.
- The subquery has only one table in the FROM clause.
- The transformation results in 15 or fewer tables in the join.
- The subquery select list has only one column, guaranteed by a unique index to have unique values.
- The comparison operator of the predicate containing the subquery is IN, = ANY, or = SOME.

- For a noncorrelated subquery, the left side of the predicate is a single column with the same data type and length as the subquery's column. (For a correlated subquery, the left side can be any expression.)

For an UPDATE or DELETE statement, or a SELECT statement that does not meet the previous conditions for transformation, DB2 does the transformation of a correlated subquery into a join if the following conditions are true:

- The transformation does not introduce redundancy.
- The subquery is correlated to its immediate outer query.
- The FROM clause of the subquery contains only one table, and the outer query (for SELECT), UPDATE, or DELETE references only one table.
- If the outer predicate is a quantified predicate with an operator of =ANY or an IN predicate, the following conditions are true:
  - The left side of the outer predicate is a single column.
  - The right side of the outer predicate is a subquery that references a single column.
  - The two columns have the same data type and length.
- The subquery does not contain the GROUP BY or DISTINCT clauses.
- The subquery does not contain column functions.
- The SELECT clause of the subquery does not contain a user-defined function with an external action or a user-defined function that modifies data.
- The subquery predicate is a Boolean term predicate.
- The predicates in the subquery that provide correlation are stage 1 predicates.
- The subquery does not contain nested subqueries.
- The subquery does not contain a self-referencing UPDATE or DELETE.
- For a SELECT statement, the query does not contain the FOR UPDATE OF clause.
- For an UPDATE or DELETE statement, the statement is a searched UPDATE or DELETE.
- For a SELECT statement, parallelism is not enabled.

For a statement with multiple subqueries, DB2 does the transformation only on the last subquery in the statement that qualifies for transformation.

Example: The following subquery can be transformed into a join because it meets the first set of conditions for transformation:

```
SELECT * FROM EMP
  WHERE DEPTNO IN
   (SELECT DEPTNO FROM DEPT
      WHERE LOCATION IN ('SAN JOSE', 'SAN FRANCISCO')
      AND DIVISION = 'MARKETING');
```

If there is a department in the marketing division which has branches in both San Jose and San Francisco, the result of the above SQL statement is not the same as if a join were done. The join makes each employee in this department appear twice because it matches once for the department of location San Jose and again of location San Francisco, although it is the same department. Therefore, it is clear that to transform a subquery into a join, the uniqueness of the subquery select list must be guaranteed. For this example, a unique index on any of the following sets of columns would guarantee uniqueness:
- (DEPTNO)
- (DIVISION, DEPTNO)

- (DEPTNO, DIVISION).

The resultant query is:

```
SELECT EMP.* FROM EMP, DEPT
    WHERE EMP.DEPTNO = DEPT.DEPTNO AND
          DEPT.LOCATION IN ('SAN JOSE', 'SAN FRANCISCO') AND
          DEPT.DIVISION = 'MARKETING';
```

Example: The following subquery can be transformed into a join because it meets the second set of conditions for transformation:

```
UPDATE T1 SET T1.C1 = 1
  WHERE T1.C1 =ANY
    (SELECT T2.C1 FROM T2
      WHERE T2.C2 = T1.C2);
```

Results from EXPLAIN: For information about the result in a plan table for a subquery that is transformed into a join operation, see "Is a subquery transformed into a join?" on page 683.

Subquery tuning

The following three queries all retrieve the same rows. All three retrieve data about all designers in departments that are responsible for projects that are part of major project MA2100. These three queries show that there are several ways to retrieve a desired result.

Query A: A join of two tables

```
SELECT DSN8710.EMP.* FROM DSN8710.EMP, DSN8710.PROJ
    WHERE  JOB = 'DESIGNER'
      AND  WORKDEPT = DEPTNO
      AND  MAJPROJ = 'MA2100';
```

Query B: A correlated subquery

```
SELECT * FROM DSN8710.EMP X
    WHERE  JOB = 'DESIGNER'
      AND  EXISTS (SELECT 1 FROM DSN8710.PROJ
                   WHERE  DEPTNO = X.WORKDEPT
                     AND  MAJPROJ = 'MA2100');
```

Query C: A noncorrelated subquery

```
SELECT * FROM DSN8710.EMP
    WHERE  JOB = 'DESIGNER'
      AND  WORKDEPT IN (SELECT DEPTNO FROM DSN8710.PROJ
                        WHERE  MAJPROJ = 'MA2100');
```

If you need columns from both tables EMP and PROJ in the output, you must use a join.

PROJ might contain duplicate values of DEPTNO in the subquery, so that an equivalent join cannot be written.

In general, query A might be the one that performs best. However, if there is no index on DEPTNO in table PROJ, then query C might perform best. The IN-subquery predicate in query C is indexable. Therefore, if an index on WORKDEPT exists, DB2 might do IN-list access on table EMP. If you decide that a join cannot be used and there is an available index on DEPTNO in table PROJ, then query B might perform best.

When looking at a problem subquery, see if the query can be rewritten into another format or see if there is an index that you can create to help improve the performance of the subquery.

It is also important to know the sequence of evaluation, for the different subquery predicates as well as for all other predicates in the query. If the subquery predicate is costly, perhaps another predicate could be evaluated before that predicate so that the rows would be rejected before even evaluating the problem subquery predicate.

Using scrollable cursors efficiently

The following recommendations help you get the best performance from your scrollable cursors:

- Determine when scrollable cursors work best for you.

  Scrollable cursors are a valuable tool for writing applications such as screen-based applications, in which the result table is small and you often move back and forth through the data. However, scrollable cursors require more DB2 processing than non-scrollable cursors. If your applications require large result tables or you only need to move sequentially forward through the data, use non-scrollable cursors.

- Declare scrollable cursors as SENSITIVE only if you need to see the latest data.

  If you do not need to see updates that are made by other cursors or application processes, using a cursor that you declare as INSENSITIVE requires less processing by DB2.

- To ensure maximum concurrency when you use a scrollable cursor for positioned update and delete operations, specify ISOLATION(CS) and CURRENTDATA(NO) when you bind packages and plans that contain updatable scrollable cursors. See "Chapter 17. Planning for concurrency" on page 323 for more details.

- Use the FETCH FIRST *n* ROWS ONLY clause with scrollable cursors when it is appropriate.

  In a distributed environment, when you need to retrieve a limited number of rows, FETCH FIRST *n* ROWS ONLY can improve your performance for distributed queries that use DRDA access by eliminating unneeded network traffic. See "Specifying FETCH FIRST n ROWS ONLY" on page 388 for more information.

  In a local environment, if you need to scroll through a limited subset of rows in a table, you can use FETCH FIRST *n* ROWS ONLY to make the result table smaller.

- In a distributed environment, if you do not need to use your scrollable cursors to modify data, do your cursor processing in a stored procedure.

  Using stored procedures can decrease the amount of network traffic that your application requires.

- Create TEMP table spaces that are large enough to process your scrollable cursors.

  See Part 2 of *DB2 Installation Guide* for information on calculating the appropriate size for declared temporary tables that you use for scrollable cursors.

- Remember to commit changes often.

  Because you often leave scrollable cursors open longer than non-scrollable cursors, it is important to commit changes often enough. Declare your scrollable cursors WITH HOLD to prevent the cursors from closing after a commit operation.

Special techniques to influence access path selection

> **ATTENTION**
>
> This section describes tactics for rewriting queries and modifying catalog statistics to influence DB2's method of selecting access paths. In a later release of DB2, the selection method might change, causing your changes to degrade performance. Save the old catalog statistics or SQL before you consider making any changes to control the choice of access path. Before and after you make any changes, take performance measurements. When you migrate to a new release, examine the performance again. Be prepared to back out any changes that have degraded performance.

This section contains the following information about determining and changing access paths:
- Obtaining information about access paths
- "Minimizing overhead for retrieving few rows: OPTIMIZE FOR n ROWS" on page 656
- "Fetching a limited number of rows: FETCH FIRST n ROWS ONLY" on page 658
- "Reducing the number of matching columns" on page 658
- "Adding extra local predicates" on page 660
- "Rearranging the order of tables in a FROM clause" on page 663
- "Updating catalog statistics" on page 663
- "Using a subsystem parameter" on page 665

Obtaining information about access paths

There are several ways to obtain information about DB2 access paths:
- Use Visual Explain

  The DB2 Visual Explain tool, which is invoked from a workstation client, can be used to display and analyze information on access paths chosen by DB2. The tool provides you with an easy-to-use interface to the PLAN_TABLE output and allows you to invoke EXPLAIN for dynamic SQL statements. You can also access the catalog statistics for certain referenced objects of an access path. In addition, the tool allows you to archive EXPLAIN output from previous SQL statements to analyze changes in your SQL environment. See *DB2 Visual Explain online help* for more information.

- Run DB2 Performance Monitor accounting reports

  Another way to track performance is with the DB2 Performance Monitor accounting reports. The accounting report, short layout, ordered by PLANNAME, lists the primary performance figures. Check the plans that contain SQL statements whose access paths you tried to influence. If the elapsed time, TCB time, or number of getpage requests increases sharply without a corresponding increase in the SQL activity, then there could be a problem. You can use DB2 PM Online Monitor to track events after your changes have been implemented, providing immediate feedback on the effects of your changes.

- Specify the bind option EXPLAIN

  You can also use the EXPLAIN option when you bind or rebind a plan or package. Compare the new plan or package for the statement to the old one. If the new one has a table space scan or a nonmatching index space scan, but the old one did not, the problem is probably the statement. Investigate any changes in access path in the new plan or package; they could represent performance improvements or degradations. If neither the accounting report ordered by

PLANNAME or PACKAGE nor the EXPLAIN statement suggest corrective action, use the DB2 PM SQL activity reports for additional information. For more information on using EXPLAIN, see "Obtaining PLAN_TABLE information from EXPLAIN" on page 668.

Minimizing overhead for retrieving few rows: OPTIMIZE FOR n ROWS

When an application executes a SELECT statement, DB2 assumes that the application will retrieve all the qualifying rows. This assumption is most appropriate for batch environments. However, for interactive SQL applications, such as SPUFI, it is common for a query to define a very large potential result set but retrieve only the first few rows. The access path that DB2 chooses might not be optimal for those interactive applications.

This section discusses the use of OPTIMIZE FOR n ROWS to affect the performance of interactive SQL applications. Unless otherwise noted, this information pertains to local applications. For more information on using OPTIMIZE FOR n ROWS in distributed applications, see "Specifying OPTIMIZE FOR n ROWS" on page 385.

What OPTIMIZE FOR n ROWS does: The OPTIMIZE FOR *n* ROWS clause lets an application declare its intent to do either of these things:

* Retrieve only a subset of the result set
* Give priority to the retrieval of the first few rows

DB2 uses the OPTIMIZE FOR *n* ROWS clause to choose access paths that minimize the response time for retrieving the first few rows. For distributed queries, the value of *n* determines the number of rows that DB2 sends to the client on each DRDA network transmission. See "Specifying OPTIMIZE FOR n ROWS" on page 385 for more information on using OPTIMIZE FOR *n* ROWS in the distributed environment.

Use OPTIMIZE FOR 1 ROW to avoid sorts: You can influence the access path most by using OPTIMIZE FOR 1 ROW. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly. This means that whenever possible, DB2 avoids any access path that involves a sort. If you specify a value for n that is anything but 1, DB2 chooses an access path based on cost, and you won't necessarily avoid sorts.

How to specify OPTIMIZE FOR n ROWS for a CLI application: For a Call Level Interface (CLI) application, you can specify that DB2 uses OPTIMIZE FOR *n* ROWS for all queries. To do that, specify the keyword OPTIMIZEFORNROWS in the initialization file. For more information, see Chapter 3 of *DB2 ODBC Guide and Reference*.

How many rows you can retrieve with OPTIMIZE FOR n ROWS: The OPTIMIZE FOR *n* ROWS clause does not prevent you from retrieving all the qualifying rows. However, if you use OPTIMIZE FOR *n* ROWS, the total elapsed time to retrieve all the qualifying rows might be significantly greater than if DB2 had optimized for the entire result set.

When OPTIMIZE FOR n ROWS is effective: OPTIMIZE FOR *n* ROWS is effective only on queries that can be performed incrementally. If the query causes DB2 to gather the whole result set before returning the first row, DB2 ignores the OPTIMIZE FOR *n* ROWS clause, as in the following situations:

- The query uses SELECT DISTINCT or a set function distinct, such as COUNT(DISTINCT C1).
- Either GROUP BY or ORDER BY is used, and there is no index that can give the ordering necessary.
- There is a column function and no GROUP BY clause.
- The query uses UNION.

Example: Suppose you query the employee table regularly to determine the employees with the highest salaries. You might use a query like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
  FROM EMP
  ORDER BY SALARY DESC;
```

An index is defined on column EMPNO, so employee records are ordered by EMPNO. If you have also defined a descending index on column SALARY, that index is likely to be very poorly clustered. To avoid many random, synchronous I/O operations, DB2 would most likely use a table space scan, then sort the rows on SALARY. This technique can cause a delay before the first qualifying rows can be returned to the application. If you add the OPTIMIZE FOR *n* ROWS clause to the statement, as shown below:

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY
  FROM EMP
  ORDER BY SALARY DESC
  OPTIMIZE FOR 20 ROWS;
```

DB2 would most likely use the SALARY index directly because you have indicated that you will probably retrieve the salaries of only the 20 most highly paid employees. This choice avoids a costly sort operation.

Effects of using OPTIMIZE FOR n ROWS:
- The join method could change. Nested loop join is the most likely choice, because it has low overhead cost and appears to be more efficient if you want to retrieve only one row.
- An index that matches the ORDER BY clause is more likely to be picked. This is because no sort would be needed for the ORDER BY.
- List prefetch is less likely to be picked.
- Sequential prefetch is less likely to be requested by DB2 because it infers that you only want to see a small number of rows.
- In a join query, the table with the columns in the ORDER BY clause is likely to be picked as the outer table if there is an index on that outer table that gives the ordering needed for the ORDER BY clause.

Recommendation: For a local query, specify OPTIMIZE FOR *n* ROWS only in applications that frequently fetch only a small percentage of the total rows in a query result set. For example, an application might read only enough rows to fill the end user's terminal screen. In cases like this, the application might read the remaining part of the query result set only rarely. For an application like this, OPTIMIZE FOR *n* ROWS can result in better performance by causing DB2 to favor SQL access paths that deliver the first n rows as fast as possible.

When you specify OPTIMIZE FOR *n* ROWS for a remote query, a small value of *n* can help limit the number of rows that flow across the network on any given transmission.

You can improve the performance for receiving a large result set through a remote query by specifying a large value of *n* in OPTIMIZE FOR *n* ROWS. When you specify a large value, DB2 attempts to send the *n* rows in multiple transmissions. For better performance when retrieving a large result set, in addition to specifying OPTIMIZE FOR *n* ROWS with a large value of *n* in your query, do not execute other SQL statements until the entire result set for the query is processed. If retrieval of data for several queries overlaps, DB2 might need to buffer result set data in the DDF address space. See "Block fetching result sets" in Part 5 (Volume 2) of *DB2 Administration Guide* for more information.

For local or remote queries, to influence the access path most, specify OPTIMIZE for 1 ROW. This value does not have a detrimental effect on distributed queries.

Fetching a limited number of rows: FETCH FIRST n ROWS ONLY

In some applications, you execute queries that can return a large number of rows, but you need only a small subset of those rows. Retrieving the entire result table from the query can be inefficient. You can specify the FETCH FIRST *n* ROWS ONLY clause in a SELECT statement to limit the number of rows in the result table of a query to *n* rows. In addition, for a distributed query that uses DRDA access, FETCH FIRST *n* ROWS ONLY, DB2 prefetches only *n* rows.

Example: Suppose that you write an application that requires information on only the 20 employees with the highest salaries. To return only the rows of the employee table for those 20 employees, you can write a query like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
  FROM EMP
  ORDER BY SALARY DESC
  FETCH FIRST 20 ROWS ONLY;
```

Interaction between OPTIMIZE FOR n ROWS and FETCH FIRST n ROWS ONLY: In general, if you specify FETCH FIRST *n* ROWS ONLY but not OPTIMIZE FOR *n* ROWS in a SELECT statement, DB2 optimizes the query as if you had specified OPTIMIZE FOR *n* ROWS. If you specify the OPTIMIZE FOR *n* ROWS and the FETCH FIRST *m* ROWS clauses, and *n<m*, DB2 optimizes the query for *n* rows. If *m<n*, DB2 optimizes for *m* rows.

Reducing the number of matching columns

Discourage the use of a poorer performing index by reducing the index's matching predicate on its leading column. Consider the example in Figure 192 on page 660, where the index that DB2 picks is less than optimal.

DB2 picks IX2 to access the data, but IX1 would be roughly 10 times quicker. The problem is that 50% of all parts from center number 3 are still in Center 3; they have not moved. Assume that there are no statistics on the correlated columns in catalog table SYSCOLDIST. Therefore, DB2 assumes that the parts from center number 3 are evenly distributed among the 50 centers.

You can get the desired access path by changing the query. To discourage the use of IX2 for this particular query, you can change the third predicate to be nonindexable.

```
  SELECT * FROM PART_HISTORY
  WHERE
      PART_TYPE = 'BB'
  AND W_FROM = 3
  AND (W_NOW = 3 + 0)        <-- PREDICATE IS MADE NONINDEXABLE
```

Now index I2 is not picked, because it has only one match column. The preferred index, I1, is picked. The third predicate is a nonindexable predicate, so an index is not used for the compound predicate.

There are many ways to make a predicate nonindexable. The recommended way is to make the add 0 to a predicate that evaluates to a numeric value or concatenate a predicate that evaluates to a character value with an empty string.

Indexable	Nonindexable
`T1.C3=T2.C4`	`(T1.C3=T2.C4 CONCAT '')`
`T1.C1=5`	`T1.C1=5+0`

These techniques do not affect the result of the query and cause only a small amount of overhead.

The preferred technique for improving the access path when a table has correlated columns is to generate catalog statistics on the correlated columns. You can do that either by running RUNSTATS or by updating catalog table SYSCOLDIST or SYSCOLDISTSTATS manually.

```
CREATE TABLE PART_HISTORY (
    PART_TYPE  CHAR(2),       IDENTIFIES THE PART TYPE
    PART_SUFFIX CHAR(10),     IDENTIFIES THE PART
    W_NOW      INTEGER,       TELLS WHERE THE PART IS
    W_FROM     INTEGER,       TELLS WHERE THE PART CAME FROM
    DEVIATIONS INTEGER,       TELLS IF ANYTHING SPECIAL WITH THIS PART
    COMMENTS        CHAR(254),
    DESCRIPTION     CHAR(254),
    DATE1           DATE,
    DATE2           DATE,
    DATE3           DATE);

CREATE UNIQUE INDEX IX1 ON PART_HISTORY
    (PART_TYPE,PART_SUFFIX,W_FROM,W_NOW);
CREATE UNIQUE INDEX IX2 ON PART_HISTORY
    (W_FROM,W_NOW,DATE1);

+-----------------------------------------------------------------------------+
|  Table statistics            |   Index statistics   IX1        IX2          |
|------------------------------+----------------------------------------------|
|  CARDF         100,000       |   FIRSTKEYCARDF      1000         50          |
|  NPAGES         10,000       |   FULLKEYCARDF    100,000    100,000          |
|                              |   CLUSTERRATIO        99%        99%          |
|                              |   NLEAF              3000       2000          |
|                              |   NLEVELS               3          3          |
|  -------------------------------------------------------------------------   |
|              column      cardinality    HIGH2KEY    LOW2KEY                  |
|              --------    -----------    --------    -------                  |
|              Part_type   1000           'ZZ'        'AA'                     |
|              w_now       50             1000        1                        |
|              w_from      50             1000        1                        |
+-----------------------------------------------------------------------------+

Q1:
SELECT * FROM PART_HISTORY   --  SELECT ALL PARTS
WHERE PART_TYPE = 'BB'    P1 --  THAT ARE 'BB' TYPES
  AND W_FROM = 3          P2 --  THAT WERE MADE IN CENTER 3
  AND W_NOW = 3           P3 --  AND ARE STILL IN CENTER 3

+-----------------------------------------------------------------------------+
|    Filter factor of these predicates.                                       |
|     P1 = 1/1000= .001                                                        |
|     P2 = 1/50  = .02                                                         |
|     P3 = 1/50  = .02                                                         |
|  -------------------------------------------------------------------------   |
|   ESTIMATED VALUES            |    WHAT REALLY HAPPENS                       |
|                  filter  data |                    filter  data             |
|   index matchcols factor rows | index  matchcols    factor  rows            |
|    ix2    2    .02*.02    40   |  ix2     2        .02*.50   1000            |
|    ix1    1    .001      100   |  ix1     1        .001       100            |
+-----------------------------------------------------------------------------+
```

Figure 192. Reducing the number of MATCHCOLS

Adding extra local predicates

Adding local predicates on columns that have no other predicates generally has the following effect on join queries.

1. The table with the extra predicates is more likely to be picked as the outer table. That is because DB2 estimates that fewer rows qualify from the table if there are more predicates. It is generally more efficient to have the table with the fewest qualifying rows as the outer table.

2. The join method is more likely to be nested loop join. This is because nested loop join is more efficient for small amounts of data, and more predicates make DB2 estimate that less data is to be retrieved.

The proper type of predicate to add is `WHERE TX.CX=TX.CX`.

This does not change the result of the query. It is valid for a column of any data type, and causes a minimal amount of overhead. However, DB2 uses only the best filter factor for any particular column. So, if TX.CX already has another equal predicate on it, adding this extra predicate has no effect. You should add the extra local predicate to a column that is not involved in a predicate already. If index-only access is possible for a table, it is generally not a good idea to add a predicate that would prevent index-only access.

Creating indexes for efficient star schemas

A *star schema* is a database design that, in its simplest form, consists of a large table called a *fact table,* and two or more smaller tables, called *dimension tables.* More complex star schemas can be created by breaking one or more of the dimension tables into multiple tables.

To access the data in a star schema, you write SELECT statements that include join operations between the fact table and the dimension tables, but no join operations between dimension tables.

DB2 uses a special join type called a *star join* if the following conditions are true:
- The table meet the conditions that are specified in "Star schema (star join)" on page 697 .
- The STARJOIN system parameter is set to ENABLE.

DB2 uses the star join type only if the STARJOIN system parameter is set to ENABLE.

You can improve the performance of star joins by your use of indexes. This section gives suggestions for choosing indexes might improve star join performance.

Recommendations for creating indexes for star schemas
Follow these recommendations to improve performance of queries that are processed using the star join technique:
- Define a multi-column index on all key columns of the fact table. Key columns are fact table columns that have corresponding dimension tables.
- If you do not have information about the way that your data is used, first try a multi-column index on the fact table that is based on the correlation of the data. Put less highly correlated columns later in the index key than more highly correlated columns. See "Determining the order of columns in an index for a star schema" on page 662 for information on deriving an index that follows this recommendation.
- As the correlation of columns in the fact table changes, reevaluate the index to determine if columns in the index should be reordered.
- Define indexes on dimension tables to improve access to those tables.
- When you have executed a number of queries and have more information about the way that the data is used, follow these recommendations:
  - Put more selective columns at the beginning of the index.

- – If a number of queries do not reference a dimension, put the column that corresponds to that dimension at the end of the index.

  When there are multiple multi-column indexes on the fact table, and none of those indexes contain all key columns, DB2 evaluates all of the indexes and uses the index that best exploits star join.

Determining the order of columns in an index for a star schema

You can use the following method to determine the order of columns in a multi-column index. The description of the method uses the following terminology:

F A fact table.

D1...Dn
>   Dimension tables.

C1...Cn
>   Key columns in the fact table. C1 is joined to dimension D1, C2 is joined to dimension D2, and so on.

cardD1...cardDn
>   Cardinality of columns C1...Cn in dimension tables D1...Dn.

cardC1...cardCn
>   Cardinality of key columns C1...Cn in fact table F.

cardCij
>   Cardinality of pairs of column values from key columns Ci and Cj in fact table F.

cardCijk
>   Cardinality of triplets of column values from key columns Ci, Cj, and Ck in fact table F.

Density
>   A measure of the correlation of key columns in the fact table. The density is calculated as follows:

>   **For a single column**
>>   cardCi/cardDi

>   **For pairs of columns**
>>   cardCij/(cardDi*cardDj)

>   **For triplets of columns**
>>   cardCijk/(cardDi*cardDj*cardDk)

S The current set of columns whose order in the index is not yet determined.

S-{Cm**}**
>   The current set of columns, excluding column Cm

Follow these steps to derive a fact table index for a star join that joins n columns of fact table F to n dimension tables D1 through Dn:

1. Define the set of columns whose index key order is to be determined as the n columns of fact table F that correspond to dimension tables. That is, S={C1,...Cn} and L=n.
2. Calculate the density of all sets of L-1 columns in S.
3. Find the lowest density. Determine which column is not in the set of columns with the lowest density. That is, find column Cm in S, such that for every Ci in S, density(S-{Cm})<density(S-{Ci}).
4. Make Cm the Lth column of the index.
5. Remove Cm from S.

6. Decrement L by 1.
7. Repeat steps 2 through 6 *n*-2 times. The remaining column after iteration *n*-2 is the first column of the index.

Example of determining column order for a fact table index: Suppose that a star schema has three dimension tables with the following cardinalities:

```
cardD1=2000
cardD2=500
cardD3=100
```

Now suppose that the cardinalities of single columns and pairs of columns in the fact table are:

```
cardC1=2000
cardC2=433
cardC3=100
cardC12=625000
cardC13=196000
cardC23=994
```

Determine the best multi-column index for this star schema.

Step 1: Calculate the density of all pairs of columns in the fact table:

```
density(C1,C2)=625000⁄(2000*500)=0.625
density(C1,C3)=196000⁄(2000*100)=0.98
density(C2,C3)=994⁄(500*100)=0.01988
```

Step 2: Find the pair of columns with the lowest density. That pair is (C2,C3). Determine which column of the fact table is not in that pair. That column is C1.

Step 3: Make column C1 the third column of the index.

Step 4: Repeat steps 1 through 3 to determine the second and first columns of the index key:

```
density(C2)=433⁄500=0.866
density(C3)=100⁄100=1.0
```

The column with the lowest density is C2. Therefore, C3 is the second column of the index. The remaining column, C2, is the first column of the index. That is, the best order for the multi-column index is C2, C3, C1.

Rearranging the order of tables in a FROM clause

The order of tables or views in the FROM CLAUSE can affect the access path. If your query performs poorly, it could be because the join sequence is inefficient. You can determine the join sequence within a query block from the PLANNO column in the PLAN_TABLE. For information on using the PLAN_TABLE, see "Chapter 26. Using EXPLAIN to improve SQL performance" on page 667. If you think that the join sequence is inefficient, try rearranging the order of the tables and views in the FROM clause to match a join sequence that might perform better. Rearranging the columns might cause DB2 to select the better join sequence.

Updating catalog statistics

If you have the proper authority, it is possible to influence access path selection by using an SQL UPDATE or INSERT statement to change statistical values in the DB2 catalog. However, this is not generally recommended except as a last resort. While updating catalog statistics can help a certain query, other queries can be

affected adversely. Also, the UPDATE statements must be repeated after RUNSTATS resets the catalog values. You should be very careful if you attempt to update statistics. .

The example shown in Figure 192 on page 660, involving this query:

```
SELECT * FROM PART_HISTORY    --   SELECT ALL PARTS
WHERE PART_TYPE = 'BB'     P1 --   THAT ARE 'BB' TYPES
  AND W_FROM = 3           P2 --   THAT WERE MADE IN CENTER 3
  AND W_NOW = 3            P3 --   AND ARE STILL IN CENTER 3
```

is a problem with data correlation. DB2 does not know that 50% of the parts that were made in Center 3 are still in Center 3. It was circumvented by making a predicate nonindexable. But suppose there are hundreds of users writing queries similar to that query. It would not be possible to have all users change their queries. In this type of situation, the best solution is to change the catalog statistics.

For the query in Figure 192 on page 660, where the correlated columns are concatenated key columns of an index, you can update the catalog statistics in one of two ways:

- Run the RUNSTATS utility, and request statistics on the correlated columns W_FROM and W_NOW. This is the preferred method. See the discussion of maintaining statistics in the catalog in Part 5 (Volume 2) of *DB2 Administration Guide* and Part 2 of *DB2 Utility Guide and Reference* for more information.
- Update the catalog statistics manually.

Updating the catalog to adjust for correlated columns: One catalog table you can update is SYSIBM.SYSCOLDIST, which gives information about the first key column or concatenated columns of an index key. Assume that because columns W_NOW and W_FROM are correlated, there are only 100 distinct values for the combination of the two columns, rather than 2500 (50 for W_FROM * 50 for W_NOW). Insert a row like this to indicate the new cardinality:

```
INSERT INTO SYSIBM.SYSCOLDIST
      (FREQUENCY, FREQUENCYF, IBMREQD,
       TBOWNER, TBNAME, NAME, COLVALUE,
       TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
 VALUES(0, -1, 'N',
       'USRT001','PART_HISTORY','W_FROM',' ',
       'C',100,X'00040003',2);
```

Because W_FROM and W_NOW are concatenated key columns of an index, you can also put this information in SYSCOLDIST using the RUNSTATS utility. See *DB2 Utility Guide and Reference* for more information.

You can also tell DB2 about the frequency of a certain combination of column values by updating SYSIBM.SYSCOLDIST. For example, you can indicate that 1% of the rows in PART_HISTORY contain the values 3 for W_FROM and 3 for W_NOW by inserting this row into SYSCOLDIST:

```
INSERT INTO SYSIBM.SYSCOLDIST
      (FREQUENCY, FREQUENCYF, STATSTIME, IBMREQD,
       TBOWNER, TBNAME, NAME, COLVALUE,
       TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
 VALUES(0, .0100, '1996-12-01-12.00.00.000000','N',
       'USRT001','PART_HISTORY','W_FROM',X'008000000030080000000003',
       'F',-1,X'00040003',2);
```

Updating the catalog for joins with table functions: Updating catalog statistics **might cause extreme performance problems** if the statistics are not updated

correctly. Monitor performance, and be prepared to reset the statistics to their original values if performance problems arise.

Using a subsystem parameter

DB2 often does a table space scan or nonmatching index scan when the data access statistics indicate that a table is small, even though matching index access is possible. This is a problem if the table is small or empty when statistics are collected, but the table is large when it is queried. In that case, the statistics are not accurate and can lead DB2 to pick an inefficient access path.

The best solution to the problem is to run RUNSTATS again after the table is populated. However, if it is not possible to do that, you can use subsystem parameter NPGTHRSH to cause DB2 to favor matching index access over a table space scan and over nonmatching index access.

To set NPGTHRSH, which is in macro DSN6SPRM, modify and run installation job DSNTIJUZ. See Part 2 of *DB2 Installation Guide* for information on how to set subsystem parameters. The value of NPGTHRSH is an integer that indicates the tables for which DB2 favors matching index access. Values of NPGTHRSH and their meanings are:

–1   DB2 favors matching index access for all tables.

0   DB2 selects the access path based on cost, and no tables qualify for special handling. This is the default.

n>=1   If data access statistics have been collected for all tables, DB2 favors matching index access for tables for which the total number of pages on which rows of the table appear (NPAGES) is less than *n*.

If data access statistics have not been collected for some tables (NPAGES=-1 for those tables), DB2 favors matching index access for tables for which NPAGES=-1 or NPAGES<*n*.

Recommendation: Before you use NPGTHRSH, be aware that in some cases, matching index access can be more costly than a table space scan or nonmatching index access. Specify a small value for NPGTHRSH (10 or less). That limits the number of tables for which DB2 favors matching index access.

Chapter 26. Using EXPLAIN to improve SQL performance

The information under this heading, up to the end of this chapter, is Product-sensitive Programming Interface and Associated Guidance Information, as defined in "Notices" on page 937.

Definitions and purpose: EXPLAIN is a monitoring tool that produces information about the following:

- A plan, package, or SQL statement when it is bound. The output appears in a table you create called PLAN_TABLE, which is also called a *plan table*. For experienced users, you can use PLAN_TABLE to give optimization hints to DB2.
- An estimated cost of executing an SQL SELECT, INSERT, UPDATE, or DELETE statement. The output appears in a table you create called DSN_STATEMNT_TABLE, which is also called a *statement table*. For more information about statement tables, see "Estimating a statement's cost" on page 711.
- User-defined functions referred to in the statement, including the specific name and schema. The output appears in a table you create called DSN_FUNCTION_TABLE, which is also called a *function table*. For more information about function tables, see "Ensuring that DB2 executes the intended user-defined function" on page 289.

Other tools: The following tools can help you tune SQL queries:

- DB2 Visual Explain

  Visual Explain is a graphical workstation feature of DB2 that provides:
    - An easy-to-understand display of a selected access path
    - Suggestions for changing an SQL statement
    - An ability to invoke EXPLAIN for dynamic SQL statements
    - An ability to provide DB2 catalog statistics for referenced objects of an access path
    - A subsystem parameter browser with keyword 'Find' capabilities

  For information on using DB2 Visual Explain, which is a separately packaged CD-ROM provided with your DB2 Version 7 license, see *DB2 Visual Explain online help*.

- DB2 Performance Monitor (PM)

  DB2 PM is a performance monitoring tool that formats performance data. DB2 PM combines information from EXPLAIN and from the DB2 catalog. It displays access paths, indexes, tables, table spaces, plans, packages, DBRMs, host variable definitions, ordering, table access and join sequences, and lock types. Output is presented in a dialog rather than as a table, making the information easy to read and understand.

- DB2 Estimator

  DB2 Estimator for Windows is an easy-to-use, stand-alone tool for estimating the performance of DB2 for OS/390 applications. You can use it to predict the performance and cost of running the applications, transactions, SQL statements, triggers, and utilities. For instance, you can use DB2 Estimator for estimating the impact of adding or dropping an index from a table, estimating the change in response time from adding processor resources, and estimating the amount of time a utility job will take to run. DB2 Estimator for Windows can be downloaded from the Web.

Chapter overview: This chapter includes the following topics:

- "Obtaining PLAN_TABLE information from EXPLAIN"
- "Estimating a statement's cost" on page 711
- "Asking questions about data access" on page 675
- "Interpreting access to a single table" on page 683
- "Interpreting access to two or more tables (join)" on page 689
- "Interpreting data prefetch" on page 701
- "Determining sort activity" on page 705
- "Processing for views and nested table expressions" on page 706

See also "Chapter 27. Parallel operations and query performance" on page 717.

Obtaining PLAN_TABLE information from EXPLAIN

The information in PLAN_TABLE can help you to:

- Design databases, indexes, and application programs
- Determine when to rebind an application
- Determine the access path chosen for a query

For each access to a single table, EXPLAIN tells you if an index access or table space scan is used. If indexes are used, EXPLAIN tells you how many indexes and index columns are used and what I/O methods are used to read the pages. For joins of tables, EXPLAIN tells you which join method and type are used, the order in which DB2 joins the tables, and when and why it sorts any rows.

The primary use of EXPLAIN is to observe the access paths for the SELECT parts of your statements. For UPDATE and DELETE WHERE CURRENT OF, and for INSERT, you receive somewhat less information in your plan table. And some accesses EXPLAIN does not describe: for example, the access to LOB values, which are stored separately from the base table, and access to parent or dependent tables needed to enforce referential constraints.

The access paths shown for the example queries in this chapter are intended only to illustrate those examples. If you execute the queries in this chapter on your system, the access paths chosen can be different.

Steps to obtain PLAN_TABLE information: Use the following overall steps to obtain information from EXPLAIN:

1. Have appropriate access to a plan table. To create the table, see "Creating PLAN_TABLE".
2. Populate the table with the information you want. For instructions, see "Populating and maintaining a plan table" on page 673.
3. Select the information you want from the table. For instructions, see "Reordering rows from a plan table" on page 674.

Creating PLAN_TABLE

Before you can use EXPLAIN, you must create a table called PLAN_TABLE to hold the results of EXPLAIN. A copy of the statements needed to create the table are in the DB2 sample library, under the member name DSNTESC. (Unless you need the information they provide, it is not necessary to create a function table or statement table to use EXPLAIN.)

Figure 193 on page 669 shows the format of a plan table. Table 74 on page 669 shows the content of each column.

Your plan table can use many formats, but use the 51-column format because it gives you the most information. If you alter an existing plan table to add new columns, specify the columns as NOT NULL WITH DEFAULT, so that default values are included for the rows already in the table. However, as you can see in Figure 193, certain columns do allow nulls. Do not specify those columns as NOT NULL WITH DEFAULT.

```
QUERYNO            INTEGER      NOT NULL      PREFETCH           CHAR(1)       NOT NULL WITH DEFAULT
QBLOCKNO           SMALLINT     NOT NULL      COLUMN_FN_EVAL     CHAR(1)       NOT NULL WITH DEFAULT
APPLNAME           CHAR(8)      NOT NULL      MIXOPSEQ           SMALLINT      NOT NULL WITH DEFAULT
PROGNAME           CHAR(8)      NOT NULL         ---------28 column format ---------
PLANNO             SMALLINT     NOT NULL      VERSION            VARCHAR(64)   NOT NULL WITH DEFAULT
METHOD             SMALLINT     NOT NULL      COLLID             CHAR(18)      NOT NULL WITH DEFAULT
CREATOR            CHAR(8)      NOT NULL         ---------30 column format ---------
TNAME              CHAR(18)     NOT NULL      ACCESS_DEGREE      SMALLINT
TABNO              SMALLINT     NOT NULL      ACCESS_PGROUP_ID   SMALLINT
ACCESSTYPE         CHAR(2)      NOT NULL      JOIN_DEGREE        SMALLINT
MATCHCOLS          SMALLINT     NOT NULL      JOIN_PGROUP_ID     SMALLINT
ACCESSCREATOR      CHAR(8)      NOT NULL         ---------34 column format ---------
ACCESSNAME         CHAR(18)     NOT NULL      SORTC_PGROUP_ID    SMALLINT
INDEXONLY          CHAR(1)      NOT NULL      SORTN_PGROUP_ID    SMALLINT
SORTN_UNIQ         CHAR(1)      NOT NULL      PARALLELISM_MODE   CHAR(1)
SORTN_JOIN         CHAR(1)      NOT NULL      MERGE_JOIN_COLS    SMALLINT
SORTN_ORDERBY      CHAR(1)      NOT NULL      CORRELATION_NAME   CHAR(18)
SORTN_GROUPBY      CHAR(1)      NOT NULL      PAGE_RANGE         CHAR(1)       NOT NULL WITH DEFAULT
SORTC_UNIQ         CHAR(1)      NOT NULL      JOIN_TYPE          CHAR(1)       NOT NULL WITH DEFAULT
SORTC_JOIN         CHAR(1)      NOT NULL      GROUP_MEMBER       CHAR(8)       NOT NULL WITH DEFAULT
SORTC_ORDERBY      CHAR(1)      NOT NULL      IBM_SERVICE_DATA   VARCHAR(254)  NOT NULL WITH DEFAULT
SORTC_GROUPBY      CHAR(1)      NOT NULL         --------43 column format ----------
TSLOCKMODE         CHAR(3)      NOT NULL      WHEN_OPTIMIZE      CHAR(1)       NOT NULL WITH DEFAULT
TIMESTAMP          CHAR(16)     NOT NULL      QBLOCK_TYPE        CHAR(6)       NOT NULL WITH DEFAULT
REMARKS            VARCHAR(254) NOT NULL      BIND_TIME          TIMESTAMP     NOT NULL WITH DEFAULT
   ---------25 column format ---------          ------46 column format ------------
                                              OPTHINT            CHAR(8)       NOT NULL WITH DEFAULT
                                              HINT_USED          CHAR(8)       NOT NULL WITH DEFAULT
                                              PRIMARY_ACCESSTYPE CHAR(1)       NOT NULL WITH DEFAULT
                                                 -------49 column format------------
                                              PARENT_QBLOCK      SMALLINT      NOT NULL WITH DEFAULT
                                              TABLE_TYPE         CHAR(1)
                                                 -------51 column format-----------
```

Figure 193. Format of PLAN_TABLE

Table 74. Descriptions of columns in PLAN_TABLE

Column Name	Description
QUERYNO	A number intended to identify the statement being explained. For a row produced by an EXPLAIN statement, specify the number in the QUERYNO clause. For a row produced by non-EXPLAIN statements, specify the number using the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE and DELETE statement syntax. Otherwise, DB2 assigns a number based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values greater than 32767 are reported as 0. Hence, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of TIMESTAMP is unique.
QBLOCKNO	The position of the query in the statement being explained (1 for the outermost query, 2 for the next query, and so forth). For better performance, DB2 might merge a query block into another query block. When that happens, the position number of the merged query block will not be in QBLOCKNO.

Table 74. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
APPLNAME	The name of the application plan for the row. Applies only to embedded EXPLAIN statements executed from a plan or to statements explained when binding a plan. Blank if not applicable.
PROGNAME	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. Blank if not applicable.
PLANNO	The number of the step in which the query indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.
METHOD	A number (0, 1, 2, 3, or 4) that indicates the join method used for the step: **0** First table accessed, continuation of previous table accessed, or not used. **1** *Nested loop* join. For each row of the present composite table, matching rows of a new table are found and joined. **2** *Merge scan* join. The present composite table and the new table are scanned in the order of the join columns, and matching rows are joined. **3** Sorts needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, a quantified predicate, or an IN predicate. This step does not access a new table. **4** *Hybrid* join. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.
CREATOR	The creator of the new table accessed in this step, blank if METHOD is 3.
TNAME	The name of a table, created or declared temporary table, materialized view, table expression, or an intermediate result table for an outer join that is accessed in this step, blank if METHOD is 3. For an outer join, this column contains the temporary table name of the work file in the form DSNWFQB(*qblockno*). Merged views show the base table names and correlation names. A materialized view is another query block with its own materialized views, tables, and so forth.
TABNO	Values are for IBM use only.
ACCESSTYPE	The method of accessing the new table: **I** By an index (identified in ACCESSCREATOR and ACCESSNAME) **I1** By a one-fetch index scan **N** By an index scan when the matching predicate contains the IN keyword **R** By a table space scan **M** By a multiple index scan (followed by MX, MI, or MU) **MX** By an index scan on the index named in ACCESSNAME **MI** By an intersection of multiple indexes **MU** By a union of multiple indexes **blank** Not applicable to the current row
MATCHCOLS	For ACCESSTYPE I, I1, N, or MX, the number of index keys used in an index scan; otherwise, 0.
ACCESSCREATOR	For ACCESSTYPE I, I1, N, or MX, the creator of the index; otherwise, blank.
ACCESSNAME	For ACCESSTYPE I, I1, N, or MX, the name of the index; otherwise, blank.
INDEXONLY	Whether access to an index alone is enough to carry out the step, or whether data too must be accessed. Y=Yes; N=No.
SORTN_UNIQ	Whether the new table is sorted to remove duplicate rows. Y=Yes; N=No.

Table 74. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
SORTN_JOIN	Whether the new table is sorted for join method 2 or 4. Y=Yes; N=No.
SORTN_ORDERBY	Whether the new table is sorted for ORDER BY. Y=Yes; N=No.
SORTN_GROUPBY	Whether the new table is sorted for GROUP BY. Y=Yes; N=No.
SORTC_UNIQ	Whether the composite table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTC_JOIN	Whether the composite table is sorted for join method 1, 2 or 4. Y=Yes; N=No.
SORTC_ORDERBY	Whether the composite table is sorted for an ORDER BY clause or a quantified predicate. Y=Yes; N=No.
SORTC_GROUPBY	Whether the composite table is sorted for a GROUP BY clause. Y=Yes; N=No.
TSLOCKMODE	An indication of the mode of lock to be acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are: **IS** Intent share lock **IX** Intent exclusive lock **S** Share lock **U** Update lock **X** Exclusive lock **SIX** Share with intent exclusive lock **N** UR isolation; no lock If the isolation cannot be determined at bind time, then the lock mode determined by the isolation at run time is shown by the following values. **NS** For UR isolation, no lock; for CS, RS, or RR, an S lock. **NIS** For UR isolation, no lock; for CS, RS, or RR, an IS lock. **NSS** For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock. **SS** For UR, CS, or RS isolation, an IS lock; for RR, an S lock. The data in this column is right justified. For example, IX appears as a blank followed by I followed by X. If the column contains a blank, then no lock is acquired.
TIMESTAMP	Usually, the time at which the row is processed, to the last .01 second. If necessary, DB2 adds .01 second to the value to ensure that rows for two successive queries have different values.
REMARKS	A field into which you can insert any character string of 254 or fewer characters.
PREFETCH	Whether data pages are to be read in advance by prefetch. S = pure sequential prefetch; L = prefetch through a page list; blank = unknown or no prefetch.
COLUMN_FN_EVAL	When an SQL column function is evaluated. R = while the data is being read from the table or index; S = while performing a sort to satisfy a GROUP BY clause; blank = after data retrieval and after any sorts.
MIXOPSEQ	The sequence number of a step in a multiple index operation. **1, 2, ... n** For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, or MU.) **0** For any other rows (ACCESSTYPE is I, I1, M, N, R, or blank.)
VERSION	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable.
COLLID	The collection ID for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable.

Note: The following nine columns, from ACCESS_DEGREE through CORRELATION_NAME, contain the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, each of them can contain null if the method it refers to does not apply.

Table 74. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
ACCESS_DEGREE	The number of parallel tasks or operations activated by a query. This value is determined at bind time; the actual number of parallel operations used at execution time could be different. This column contains 0 if there is a host variable.
ACCESS_PGROUP_ID	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time.
JOIN_DEGREE	The number of parallel operations or tasks used in joining the composite table with the new table. This value is determined at bind time and can be 0 if there is a host variable. The actual number of parallel operations or tasks used at execution time could be different.
JOIN_PGROUP_ID	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time.
SORTC_PGROUP_ID	The parallel group identifier for the parallel sort of the composite table.
SORTN_PGROUP_ID	The parallel group identifier for the parallel sort of the new table.
PARALLELISM_MODE	The kind of parallelism, if any, that is used at bind time: **I** Query I/O parallelism **C** Query CP parallelism **X** Sysplex query parallelism
MERGE_JOIN_COLS	The number of columns that are joined during a merge scan join (Method=2).
CORRELATION_NAME	The correlation name of a table or view that is specified in the statement. If there is no correlation name, then the column is blank.
PAGE_RANGE	Whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. Y = Yes; blank = No.
JOIN_TYPE	The type of join: **F** FULL OUTER JOIN **L** LEFT OUTER JOIN **S** STAR JOIN **blank** INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
IBM_SERVICE_DATA	Values are for IBM use only.
WHEN_OPTIMIZE	When the access path was determined: **blank** At bind time, using a default filter factor for any host variables, parameter markers, or special registers. **B** At bind time, using a default filter factor for any host variables, parameter markers, or special registers; however, the statement is reoptimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(VARS) must be specified for reoptimization to occur. **R** At run time, using input variables for any host variables, parameter markers, or special registers. The bind option REOPT(VARS) must be specified for this to occur.

Table 74. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
QBLOCK_TYPE	For each query block, an indication of the type of SQL operation performed. For the outermost query, this column identifies the statement type. Possible values: **SELECT** — SELECT **INSERT** — INSERT **UPDATE** — UPDATE **DELETE** — DELETE **SELUPD** — SELECT with FOR UPDATE OF **DELCUR** — DELETE WHERE CURRENT OF CURSOR **UPDCUR** — UPDATE WHERE CURRENT OF CURSOR **CORSUB** — Correlated subquery **NCOSUB** — Noncorrelated subquery **TABLEX** — Table expression **UNION** — UNION **UNIONA** — UNION ALL
BIND_TIME	The time at which the plan or package for this statement or query block was bound. For static SQL statements, this is a full-precision timestamp value. For dynamic SQL statements, this is the value contained in the TIMESTAMP column of PLAN_TABLE appended by 4 zeroes.
OPTHINT	A string that you use to identify this row as an optimization hint for DB2. DB2 uses this row as input when choosing an access path.
HINT_USED	If DB2 used one of your optimization hints, it puts the identifier for that hint (the value in OPTHINT) in this column.
PRIMARY_ACCESSTYPE	Indicates whether direct row access will be attempted first: **D** — DB2 will try to use direct row access. If DB2 cannot use direct row access at runtime, it uses the access path described in the ACCESSTYPE column of PLAN_TABLE. **blank** — DB2 will not try to use direct row access.
PARENT_QBLOCK	A number that indicates the QBLOCKNO of the parent query block.
TABLE_TYPE	The type of new table: **F** — Table function **Q** — Temporary intermediate result table (not materialized) **T** — Table **W** — Work file The value of the column is null if the query uses GROUP BY, ORDER BY, or DISTINCT, which requires an implicit sort.

Populating and maintaining a plan table

For the two distinct ways to populate a plan table, see:

- "Executing the SQL statement EXPLAIN" on page 674
- "Binding with the option EXPLAIN(YES)" on page 674

When you populate the plan table through DB2's EXPLAIN, any INSERT triggers on the table are not activated. If you insert rows yourself, then those triggers are activated.

For tips on maintaining a growing plan table, see "Maintaining a plan table" on page 674.

Executing the SQL statement EXPLAIN

You can populate PLAN_TABLE by executing the SQL statement EXPLAIN. In the statement, specify a single explainable SQL statement in the FOR clause.

You can execute EXPLAIN either statically from an application program, or dynamically, using QMF or SPUFI. For instructions and for details of the authorization you need on PLAN_TABLE, see *DB2 SQL Reference*.

Binding with the option EXPLAIN(YES)

You can populate a plan table when you bind or rebind a plan or package. Specify the option EXPLAIN(YES). EXPLAIN obtains information about the access paths for all explainable SQL statements in a package or the DBRMs of a plan. The information appears in table *package_owner*.PLAN_TABLE or *plan_owner*.PLAN_TABLE. For dynamically prepared SQL, the qualifier of PLAN_TABLE is the current SQLID.

Performance considerations: EXPLAIN as a bind option should not be a performance concern. The same processing for access path selection is performed, regardless of whether you use EXPLAIN(YES) or EXPLAIN (NO). With EXPLAIN(YES), there is only a small amount of overhead processing to put the results in a plan table.

If a plan or package that was previously bound with EXPLAIN(YES) is automatically rebound, the value of field EXPLAIN PROCESSING on installation panel DSNTIPO determines whether EXPLAIN is run again during the automatic rebind. Again, there is a small amount of overhead for inserting the results into a plan table.

EXPLAIN for remote binds: A remote requester that accesses DB2 can specify EXPLAIN(YES) when binding a package at the DB2 server. The information appears in a plan table at the server, not at the requester. If the requester does not support the propagation of the option EXPLAIN(YES), rebind the package at the requester with that option to obtain access path information. You cannot get information about access paths for SQL statements that use private protocol.

Maintaining a plan table

DB2 adds rows to PLAN_TABLE as you choose; it does not automatically delete rows. To clear the table of obsolete rows, use DELETE, just as you would for deleting rows from any table. You can also use DROP TABLE to drop a plan table completely.

Reordering rows from a plan table

Several processes can insert rows into the same plan table. To understand access paths, you must retrieve the rows for a particular query in an appropriate order.

Retrieving rows for a plan

The rows for a particular plan are identified by the value of APPLNAME. The following query to a plan table returns the rows for all the explainable statements in a plan in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
   WHERE APPLNAME = 'APPL1'
   ORDER BY TIMESTAMP, QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

The result of the ORDER BY clause shows whether there are:
* Multiple QBLOCKNOs within a QUERYNO
* Multiple PLANNOs within a QBLOCKNO
* Multiple MIXOPSEQs within a PLANNO

All rows with the same non-zero value for QBLOCKNO and the same value for QUERYNO relate to a step within the query. QBLOCKNOs are not necessarily executed in the order shown in PLAN_TABLE. But within a QBLOCKNO, the PLANNO column gives the substeps in the order they execute.

For each substep, the TNAME column identifies the table accessed. Sorts can be shown as part of a table access or as a separate step.

What if QUERYNO=0? In a program with more than 32767 lines, all values of QUERYNO greater than 32767 are reported as 0. For entries containing QUERYNO=0, use the timestamp, which is guaranteed to be unique, to distinguish individual statements.

Retrieving rows for a package

The rows for a particular package are identified by the values of PROGNAME, COLLID, and VERSION. Those columns correspond to the following four-part naming convention for packages:

```
LOCATION.COLLECTION.PACKAGE_ID.VERSION
```

COLLID gives the COLLECTION name, and PROGNAME gives the PACKAGE_ID. The following query to a plan table return the rows for all the explainable statements in a package in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
    WHERE PROGNAME = 'PACK1' AND COLLID = 'COLL1' AND VERSION =  'PROD1'
    ORDER BY QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

Asking questions about data access

When you examine your EXPLAIN results, try to answer the following questions:
- "Is access through an index? (ACCESSTYPE is I, I1, N or MX)"
- "Is access through more than one index? (ACCESSTYPE=M)" on page 676
- "How many columns of the index are used in matching? (MATCHCOLS=n)" on page 677
- "Is the query satisfied using only the index? (INDEXONLY=Y)" on page 677
- "Is direct row access possible? (PRIMARY_ACCESSTYPE = D)" on page 677
- "Is a view or nested table expression materialized?" on page 681
- "Was a scan limited to certain partitions? (PAGE_RANGE=Y)" on page 681
- "What kind of prefetching is done? (PREFETCH = L, S, or blank)" on page 682
- "Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)" on page 682
- "Are sorts performed?" on page 682
- "Is a subquery transformed into a join?" on page 683
- "When are column functions evaluated? (COLUMN_FN_EVAL)" on page 683

As explained in this section, they can be answered in terms of values in columns of a plan table.

Is access through an index? (ACCESSTYPE is I, I1, N or MX)

If the column ACCESSTYPE in the plan table has one of those values, DB2 uses an index to access the table named in column TNAME. The columns ACCESSCREATOR and ACCESSNAME identify the index. For a description of methods of using indexes, see "Index access paths" on page 685.

Is access through more than one index? (ACCESSTYPE=M)

Those values indicate that DB2 uses a set of indexes to access a single table. A set of rows in the plan table contain information about the multiple index access. The rows are numbered in column MIXOPSEQ in the order of execution of steps in the multiple index access. (If you retrieve the rows in order by MIXOPSEQ, the result is similar to postfix arithmetic notation.)

The examples in Figure 194 and Figure 195 have these indexes: IX1 on T(C1) and IX2 on T(C2). DB2 processes the query in the following steps:

1. Retrieve all the qualifying record identifiers (RIDs) where C1=1, using index IX1.
2. Retrieve all the qualifying RIDs where C2=1, using index IX2. The intersection of those lists is the final set of RIDs.
3. Access the data pages needed to retrieve the qualified rows using the final RID list.

```
SELECT * FROM T
   WHERE C1 = 1 AND C2 = 1;
```

TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	PREFETCH	MIXOP-SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX2	Y		2
T	MI	0		N		3

Figure 194. PLAN_TABLE output for example with intersection (AND) operator

The same index can be used more than once in a multiple index access, because more than one predicate could be matching, as in Figure 195.

```
SELECT * FROM T
   WHERE C1 BETWEEN 100 AND 199 OR
         C1 BETWEEN 500 AND 599;
```

TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	PREFETCH	MIXOP-SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX1	Y		2
T	MU	0		N		3

Figure 195. PLAN_TABLE output for example with union (OR) operator

DB2 processes the query in the following steps:

1. Retrieve all RIDs where C1 is between 100 and 199, using index IX1.
2. Retrieve all RIDs where C1 is between 500 and 599, again using IX1. The union of those lists is the final set of RIDs.
3. Retrieve the qualified rows using the final RID list.

How many columns of the index are used in matching? (MATCHCOLS=n)

If MATCHCOLS is 0, the access method is called a *nonmatching index scan*. All the index keys and their RIDs are read.

If MATCHCOLS is greater than 0, the access method is called a *matching index scan*: the query uses predicates that match the index columns.

In general, the matching predicates on the leading index columns are equal or IN predicates. The predicate that matches the final index column can be an equal, IN, or range predicate (<, <=, >, >=, LIKE, or BETWEEN).

The following example illustrates matching predicates:

```
SELECT * FROM EMP
  WHERE JOBCODE = '5' AND SALARY > 60000 AND LOCATION = 'CA';

 INDEX XEMP5 on (JOBCODE, LOCATION, SALARY, AGE);
```

The index XEMP5 is the chosen access path for this query, with MATCHCOLS = 3. Two equal predicates are on the first two columns and a range predicate is on the third column. Though the index has four columns in the index, only three of them can be considered matching columns.

Is the query satisfied using only the index? (INDEXONLY=Y)

In this case, the method is called *index-only access*. For a SELECT operation, all the columns needed for the query can be found in the index and DB2 does not access the table. For an UPDATE or DELETE operation, only the index is required to read the selected row.

Index-only access to data is not possible for any step that uses list prefetch, which is described under "What kind of prefetching is done? (PREFETCH = L, S, or blank)" on page 682. Index-only access is not possible when returning varying-length data in the result set or a VARCHAR column has a LIKE predicate, unless the VARCHAR FROM INDEX field of installation panel DSNTIP4 is set to YES and plan or packages have been rebound to pick up the change. See Part 2 of *DB2 Installation Guide* for more information.

If access is by more than one index, INDEXONLY is Y for a step with access type MX, because the data pages are not actually accessed until all the steps for intersection (MI) or union (MU) take place.

When an SQL application uses index-only access for a ROWID column, the application claims the table space or table space partition. As a result, contention may occur between the SQL application and a utility that drains the table space or partition. Index-only access to a table for a ROWID column is not possible if the associated table space or partition is in an incompatible restrictive state. For example, an SQL application can make a read claim on the table space only if the restrictive state allows readers.

Is direct row access possible? (PRIMARY_ACCESSTYPE = D)

If an application selects a row from a table that contains a ROWID column, the row ID value implicitly contains the location of the row. If you use that row ID value in

the search condition of subsequent SELECTs, DELETEs, or UPDATEs, DB2 might be able to navigate directly to the row. This access method is called *direct row access*.

Direct row access is very fast, because DB2 does not need to use the index or a table space scan to find the row. Direct row access can be used on any table that has a ROWID column.

To use direct row access, you first select the values of a row into host variables. The value that is selected from the ROWID column contains the location of that row. Later, when you perform queries that access that row, you include the row ID value in the search condition. If DB2 determines that it can use direct row access, it uses the row ID value to navigate directly to the row. See "Example: Coding with row IDs for direct row access" on page 679 for a coding example.

Which predicates qualify for direct row access?

For a query to qualify for direct row access, the search condition must be a Boolean term *stage 1* predicate that fits one of these descriptions:

1. A simple Boolean term predicate of the form COL=*noncolumn expression*, where COL has the ROWID data type and *noncolumn expression* contains a row ID
2. A simple Boolean term predicate of the form COL IN *list*, where COL has the ROWID data type and the values in *list* are row IDs, and an index is defined on COL
3. A compound Boolean term that combines several simple predicates using the AND operator, and one of the simple predicates fits description 1 or 2

However, just because a query qualifies for direct row access does not mean that that access path is always chosen. If DB2 determines that another access path is better, direct row access is not chosen.

Examples: In the following predicate example, ID is a ROWID column in table T1. A unique index exists on that ID column. The host variables are of the ROWID type.

```
WHERE ID IN (:hv_rowid1,:hv_rowid2,:hv_rowid3)
```

The following predicate also qualifies for direct row access:

```
WHERE ID = ROWID(X'F0DFD230E3C0D80D81C201AA0A280100000000000203')
```

Searching for propagated rows: If rows are propagated from one table to another, do not expect to use the same row ID value from the source table to search for the same row in the target table, or vice versa. This does not work when direct row access is the access path chosen. For example, assume that the host variable below contains a row ID from SOURCE:

```
SELECT * FROM TARGET
  WHERE ID = :hv_rowid
```

Because the row ID location is not the same as in the source table, DB2 will most likely not find that row. Search on another column to retrieve the row you want.

Reverting to ACCESSTYPE

Although DB2 might plan to use direct row access, circumstances can cause DB2 to not use direct row access at run time. DB2 remembers the location of the row as of the time it is accessed. However, that row can change locations (such as after a REORG) between the first and second time it is accessed, which means that DB2

cannot use direct row access to find the row on the second access attempt. Instead of using direct row access, DB2 uses the access path that is shown in the ACCESSTYPE column of PLAN_TABLE.

If the predicate you are using to do direct row access is not indexable and if DB2 is unable to use direct row access, then DB2 uses a table space scan to find the row. This can have a profound impact on the performance of applications that rely on direct row access. Write your applications to handle the possibility that direct row access might not be used. Some options are to:

- Ensure that your application does not try to remember ROWID columns across reorganizations of the table space.

  When your application commits, it releases its claim on the table space; it is possible that a REORG can run and move the row, which disables direct row access. Plan your commit processing accordingly; use the returned row ID value before committing, or re-select the row ID value after a commit is issued.

  If you are storing ROWID columns from another table, update those values after the table with the ROWID column is reorganized.

- Create an index on the ROWID column, so that DB2 can use the index if direct row access is disabled.

- Supplement the ROWID column predicate with another predicate that enables DB2 to use an existing index on the table. For example, after reading a row, an application might perform the following update:

```
EXEC SQL UPDATE EMP
SET SALARY = :hv_salary + 1200
 WHERE EMP_ROWID = :hv_emp_rowid
 AND EMPNO = :hv_empno;
```

  If an index exists on EMPNO, DB2 can use index access if direct access fails. The additional predicate ensures DB2 does not revert to a table space scan.

Using direct row access and other access methods

Parallelism: Direct row access and parallelism are mutually exclusive. If a query qualifies for both direct row access and parallelism, direct row access is used. If direct row access fails, DB2 does not revert to parallelism; instead it reverts to the backup access type (as designated by column ACCESSTYPE in the PLAN_TABLE). This might result in a table space scan. To avoid a table space scan in case direct row access fails, add an indexed column to the predicate.

RID list processing: Direct row access and RID list processing are mutually exclusive. If a query qualifies for both direct row access and RID list processing, direct row access is used. If direct row access fails, DB2 does not revert to RID list processing; instead it reverts to the backup access type.

Example: Coding with row IDs for direct row access

Figure 196 on page 680 is a portion of a C program that shows you how to obtain the row ID value for a row, and then to use that value to find the row efficiently when you want to modify it.

```
/*************************/
/* Declare host variables */
/*************************/
EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE IS BLOB_LOCATOR hv_picture;
   SQL TYPE IS CLOB_LOCATOR hv_resume;
   SQL TYPE IS ROWID hv_emp_rowid;
   short hv_dept, hv_id;
   char hv_name[30];
   decimal hv_salary[5,2];
EXEC SQL END DECLARE SECTION;

/*********************************************************/
/* Retrieve the picture and resume from the PIC_RES table */
/*********************************************************/
strcpy(hv_name, "Jones");
EXEC SQL SELECT PR.PICTURE, PR.RESUME INTO :hv_picture, :hv_resume
  FROM PIC_RES PR
  WHERE PR.Name = :hv_name;
```

Figure 196. Example of using a row ID value for direct row access (Part 1 of 4)

```
/*********************************************************/
/* Insert a row into the EMPDATA table that contains the  */
/* picture and resume you obtained from the PIC_RES table */
/*********************************************************/
EXEC SQL INSERT INTO EMPDATA
  VALUES (DEFAULT,9999,'Jones', 35000.00, 99,
  :hv_picture, :hv_resume);

/*********************************************************/
/* Now retrieve some information about that row,          */
/* including the ROWID value.                             */
/*********************************************************/
hv_dept = 99;
EXEC SQL SELECT E.SALARY,  E.EMP_ROWID
  INTO :hv_salary, :hv_emp_rowid
  FROM EMPDATA E
  WHERE E.DEPTNUM = :hv_dept AND E.NAME = :hv_name;
```

Figure 196. Example of using a row ID value for direct row access (Part 2 of 4)

```
/*********************************************************/
/* Update columns SALARY, PICTURE, and RESUME.  Use the  */
/* ROWID value you obtained in the previous statement    */
/* to access the row you want to update.                 */
/* smiley_face and update_resume are                     */
/* user-defined functions that are not shown here.       */
/*********************************************************/
EXEC SQL UPDATE EMPDATA
  SET SALARY = :hv_salary + 1200,
  PICTURE = smiley_face(:hv_picture),
  RESUME = update_resume(:hv_resume)
  WHERE EMP_ROWID = :hv_emp_rowid;
```

Figure 196. Example of using a row ID value for direct row access (Part 3 of 4)

```
/**********************************************************/
/* Use the ROWID value to obtain the employee ID from the */
/* same record.                                           */
/**********************************************************/
EXEC SQL SELECT E.ID INTO :hv_id
  FROM EMPDATA E
  WHERE E.EMP_ROWID = :hv_emp_rowid;

/**********************************************************/
/* Use the ROWID value to delete the employee record     */
/* from the table.                                        */
/**********************************************************/
EXEC SQL DELETE FROM EMPDATA
  WHERE EMP_ROWID = :hv_emp_rowid;
```

Figure 196. Example of using a row ID value for direct row access (Part 4 of 4)

Is a view or nested table expression materialized?

When the column TNAME names a view or nested table expression (NTE), it indicates that the view or nested table expression is materialized. *Materialization* means that the data rows selected by the view or NTE are put into a work file to be processed like a table. If a view or nested table expression is materialized, that step appears in the plan table with a separate value of QBLOCKNO and the name of the view or nested table expression in TNAME. When DB2 can process the view or nested table expression by referring only to the base table, there is no view or nested table expression name in the column TNAME. (For a more detailed description of materialization, see "Processing for views and nested table expressions" on page 706.)

Was a scan limited to certain partitions? (PAGE_RANGE=Y)

DB2 can limit a scan of data in a partitioned table space to one or more partitions. The method is called a *limited partition scan*. The query must provide a predicate on the first key column of the partitioning index. Only the first key column is significant for limiting the range of the partition scan.

A limited partition scan can be combined with other access methods. For example, consider the following query:

```
SELECT .. FROM T
   WHERE (C1 BETWEEN '2002' AND '3280'
   OR C1 BETWEEN '6000' AND '8000')
   AND C2 = '6';
```

Assume that table T has a partitioned index on column C1 and that values of C1 between 2002 and 3280 all appear in partitions 3 and 4 and the values between 6000 and 8000 appear in partitions 8 and 9. Assume also that T has another index on column C2. DB2 could choose any of these access methods:

- A matching index scan on column C1. The scan reads index values and data only from partitions 3, 4, 8, and 9. (PAGE_RANGE=N)

- A matching index scan on column C2. (DB2 might choose that if few rows have C2=6.) The matching index scan reads all RIDs for C2=6 from the index on C2 and corresponding data pages from partitions 3, 4, 8, and 9. (PAGE_RANGE=Y)

- A table space scan on T. DB2 avoids reading data pages from any partitions except 3, 4, 8 and 9. (PAGE_RANGE=Y)

Joins: Limited partition scan can be used for each table accessed in a join.

Restrictions: Limited partition scan is not supported when host variables or parameter markers are used on the first key of the primary index. This is because the qualified partition range based on such a predicate is unknown at bind time. If you think you can benefit from limited partition scan but you have host variables or parameter markers, consider binding with REOPT(VARS).

If you have predicates using an OR operator and one of the predicates refers to a column of the partitioning index that is not the first key column of the index, then DB2 does not use limited partition scan.

What kind of prefetching is done? (PREFETCH = L, S, or blank)

Prefetching is a method of determining in advance that a set of data pages is about to be used and then reading the entire set into a buffer with a single asynchronous I/O operation. If the value of PREFETCH is:

* S, the method is called *sequential prefetch*. The data pages that are read in advance are sequential. A table space scan always uses sequential prefetch. An index scan might not use it. For a more complete description, see "Sequential prefetch (PREFETCH=S)" on page 701.
* L, the method is called *list prefetch*. One or more indexes are used to select the RIDs for a list of data pages to be read in advance; the pages need not be sequential. Usually, the RIDs are sorted. The exception is the case of a hybrid join (described under "Hybrid join (METHOD=4)" on page 695) when the value of column SORTN_JOIN is N. For a more complete description, see "List prefetch (PREFETCH=L)" on page 702.
* Blank, prefetching is not chosen as an access method. However, depending on the pattern of the page access, data can be prefetched at execution time through a process called *sequential detection*. For a description of that process, see "Sequential detection at execution time" on page 703.

Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)

Parallel processing applies only to read-only queries.

If mode is:	DB2 plans to use:
I	Parallel I/O operations
C	Parallel CP operations
X	Sysplex query parallelism

Non-null values in columns ACCESS_DEGREE and JOIN_DEGREE indicate to what degree DB2 plans to use parallel operations. At execution time, however, DB2 might not actually use parallelism, or it might use fewer operations in parallel than were originally planned. For a more complete description , see "Chapter 27. Parallel operations and query performance" on page 717. For more information about Sysplex query parallelism, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

Are sorts performed?

SORTN_JOIN and SORTC_JOIN: SORTN_JOIN indicates that the new table of a join is sorted before the join. (For hybrid join, this is a sort of the RID list.) When SORTN_JOIN and SORTC_JOIN are both 'Y', two sorts are performed for the join. The sorts for joins are indicated on the same row as the new table access.

METHOD 3 sorts: These are used for ORDER BY, GROUP BY, SELECT DISTINCT, UNION, or a quantified predicate. A quantified predicate is 'col = ANY (fullselect)' or 'col = SOME (fullselect)' . They are indicated on a separate row. A single row of the plan table can indicate two sorts of a composite table, but only one sort is actually done.

SORTC_UNIQ and SORTC_ORDERBY: SORTC_UNIQ indicates a sort to remove duplicates, as might be needed by a SELECT statement with DISTINCT or UNION. SORTC_ORDERBY usually indicates a sort for an ORDER BY clause. But SORTC_UNIQ and SORTC_ORDERBY also indicate when the results of a noncorrelated subquery are sorted, both to remove duplicates and to order the results. One sort does both the removal and the ordering.

Is a subquery transformed into a join?

For better access paths, DB2 sometimes transforms subqueries into joins, as described in "Subquery transformation into join" on page 651. A plan table shows that a subquery is transformed into a join by the value in column QBLOCKNO.

- If the subquery is not transformed into a join, that means it is executed in a separate operation, and its value of QBLOCKNO is greater than the value for the outer query.
- If the subquery is transformed into a join, it and the outer query have the same value of QBLOCKNO. A join is also indicated by a value of 1, 2, or 4 in column METHOD.

When are column functions evaluated? (COLUMN_FN_EVAL)

When the column functions are evaluated is based on the access path chosen for the SQL statement.

- If the ACCESSTYPE column is I1, then a MAX or MIN function can be evaluated by one access of the index named in ACCESSNAME.
- For other values of ACCESSTYPE, the COLUMN_FN_EVAL column tells when DB2 is evaluating the column functions.

Value	Functions are evaluated ...
S	While performing a sort to satisfy a GROUP BY clause
R	While the data is being read from the table or index
blank	After data retrieval and after any sorts

Generally, values of R and S are considered better for performance than a blank.

Use variance and standard deviation with care: The VARIANCE and STDDEV functions are always evaluated late (that is, COLUMN_FN_EVAL is blank). This causes other functions in the same query block to be evaluated late as well. For example, in the following query, the sum function is evaluated later than it would be if the variance function was not present:

```
SELECT SUM(C1), VARIANCE(C1) FROM T1;
```

Interpreting access to a single table

The following sections describe different access paths that values in a plan table can indicate, along with suggestions for supplying better access paths for DB2 to choose from.

- Table space scans (ACCESSTYPE=R PREFETCH=S)
- "Index access paths" on page 685
- "UPDATE using an index" on page 689

Table space scans (ACCESSTYPE=R PREFETCH=S)

Table space scan is most often used for one of the following reasons:

- Access is through a created temporary table. (Index access is not possible for created temporary tables.)
- A matching index scan is not possible because an index is not available, or no predicates match the index columns.
- A high percentage of the rows in the table is returned. In this case, an index is not really useful because most rows need to be read anyway.
- The indexes that have matching predicates have low cluster ratios and are therefore efficient only for small amounts of data.

Assume that table T has no index on C1. The following is an example that uses a table space scan:

```
SELECT * FROM T WHERE C1 = VALUE;
```

In this case, at least every row in T must be examined to determine whether the value of C1 matches the given value.

Table space scans of nonsegmented table spaces

DB2 reads and examines every page in the table space, regardless of which table the page belongs to. It might also read pages that have been left as free space and space not yet reclaimed after deleting data.

Table space scans of segmented table spaces

If the table space is segmented, DB2 first determines which segments need to be read. It then reads only the segments in the table space that contain rows of T. If the prefetch quantity, which is determined by the size of your buffer pool, is greater than the SEGSIZE and if the segments for T are not contiguous, DB2 might read unnecessary pages. Use a SEGSIZE value that is as large as possible, consistent with the size of the data. A large SEGSIZE value is best to maintain clustering of data rows. For very small tables, specify a SEGSIZE value that is equal to the number of pages required for the table.

Recommendation for SEGSIZE value: Table 75 summarizes the recommendations for SEGSIZE, depending on how large the table is.

Table 75. Recommendations for SEGSIZE

Number of pages	SEGSIZE recommendation
≤ 28	4 to 28
> 28 < 128 pages	32
≥ 128 pages	64

Table space scans of partitioned table spaces

Partitioned table spaces are nonsegmented. A table space scan on a partitioned table space is more efficient than on a nonpartitioned table space. DB2 takes advantage of the partitions by a limited partition scan, as described under "Was a scan limited to certain partitions? (PAGE_RANGE=Y)" on page 681.

Table space scans and sequential prefetch

Regardless of the type of table space, DB2 plans to use sequential prefetch for a table space scan. For a segmented table space, DB2 might not actually use sequential prefetch at execution time if it can determine that fewer than four data pages need to be accessed. For guidance on monitoring sequential prefetch, see "Sequential prefetch (PREFETCH=S)" on page 701.

If you do not want to use sequential prefetch for a particular query, consider adding to it the clause OPTIMIZE FOR 1 ROW.

Index access paths

DB2 uses the following index access paths:
- "Matching index scan (MATCHCOLS>0)"
- "Index screening" on page 686
- "Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)" on page 686
- "IN-list index scan (ACCESSTYPE=N)" on page 686
- "Multiple index access (ACCESSTYPE is M, MX, MI, or MU)" on page 687
- "One-fetch access (ACCESSTYPE=I1)" on page 688
- "Index-only access (INDEXONLY=Y)" on page 688
- "Equal unique index (MATCHCOLS=number of index columns)" on page 689

Matching index scan (MATCHCOLS>0)

In a *matching index scan*, predicates are specified on either the leading or all of the index key columns. These predicates provide *filtering*; only specific index pages and data pages need to be accessed. If the degree of filtering is high, the matching index scan is efficient.

In the general case, the rules for determining the number of matching columns are simple, although there are a few exceptions.
- Look at the index columns from leading to trailing. For each index column, search for an indexable boolean term predicate on that column. (See "Properties of predicates" on page 624 for a definition of boolean term.) If such a predicate is found, then it can be used as a matching predicate.

  Column MATCHCOLS in a plan table shows how many of the index columns are matched by predicates.
- If no matching predicate is found for a column, the search for matching predicates stops.
- If a matching predicate is a range predicate, then there can be no more matching columns. For example, in the matching index scan example that follows, the range predicate C2>1 prevents the search for additional matching columns.
- For star joins, a missing key predicate does not cause termination of matching columns that are to be used on the fact table index.

The exceptional cases are:
- At most one IN-list predicate can be a matching predicate on an index.
- For MX accesses and index access with list prefetch, IN-list predicates cannot be used as matching predicates.
- Join predicates cannot qualify as matching predicates when doing a merge join (METHOD=2). For example, T1.C1=T2.C1 cannot be a matching predicate when doing a merge join, although any local predicates, such as C1='5' can be used.

  Join predicates can be used as matching predicates on the inner table of a nested loop join or hybrid join.

Matching index scan example: Assume there is an index on T(C1,C2,C3,C4):

```
SELECT * FROM T
  WHERE C1=1 AND C2>1
  AND C3=1;
```

Two matching columns occur in this example. The first one comes from the predicate C1=1, and the second one comes from C2>1. The range predicate on C2 prevents C3 from becoming a matching column.

Index screening

In *index screening*, predicates are specified on index key columns but are not part of the matching columns. Those predicates improve the index access by reducing the number of rows that qualify while searching the index. For example, with an index on T(C1,C2,C3,C4) in the following SQL statement, C3>0 and C4=2 are index screening predicates.

```
SELECT * FROM T
  WHERE C1 = 1
    AND C3 > 0 AND C4 = 2
    AND C5 = 8;
```

The predicates can be applied on the index, but they are not matching predicates. C5=8 is not an index screening predicate, and it must be evaluated when data is retrieved. The value of MATCHCOLS in the plan table is 1.

EXPLAIN does not directly tell when an index is screened; however, if MATCHCOLS is less than the number of index key columns, it indicates that index screening is possible.

Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)

In a *nonmatching index scan* no matching columns are in the index. Hence, all the index keys must be examined.

Because a nonmatching index usually provides no filtering, only a few cases provide an efficient access path. The following situations are examples:

- When index screening predicates exist

  In that case, not all of the data pages are accessed.
- When the clause OPTIMIZE FOR n ROWS is used

  That clause can sometimes favor a nonmatching index, especially if the index gives the ordering of the ORDER BY clause.
- When more than one table exists in a nonsegmented table space

  In that case, a table space scan reads irrelevant rows. By accessing the rows through the nonmatching index, fewer rows are read.

IN-list index scan (ACCESSTYPE=N)

An *IN-list index scan* is a special case of the matching index scan, in which a single indexable IN predicate is used as a matching equal predicate.

You can regard the IN-list index scan as a series of matching index scans with the values in the IN predicate being used for each matching index scan. The following example has an index on (C1,C2,C3,C4) and might use an IN-list index scan:

```
SELECT * FROM T
  WHERE C1=1 AND C2 IN (1,2,3)
    AND C3>0 AND C4<100;
```

The plan table shows MATCHCOLS = 3 and ACCESSTYPE = N. The IN-list scan is performed as the following three matching index scans:

```
 (C1=1,C2=1,C3>0), (C1=1,C2=2,C3>0), (C1=1,C2=3,C3>0)
```

Parallelism is supported for queries that involve IN-list index access. These queries used to run sequentially in previous releases of DB2, although parallelism could have been used when the IN-list access was for the inner table of a parallel group.

Now, in environments in which parallelism is enabled, you can see a reduction in elapsed time for queries that involve IN-list index access for the outer table of a parallel group.

Multiple index access (ACCESSTYPE is M, MX, MI, or MU)

Multiple index access uses more than one index to access a table. It is a good access path when:

- No single index provides efficient access.
- A combination of index accesses provides efficient access.

RID lists are constructed for each of the indexes involved. The unions or intersections of the RID lists produce a final list of qualified RIDs that is used to retrieve the result rows, using list prefetch. You can consider multiple index access as an extension to list prefetch with more complex RID retrieval operations in its first phase. The complex operators are union and intersection.

DB2 chooses multiple index access for the following query:

```
SELECT * FROM EMP
    WHERE (AGE = 34) OR
          (AGE = 40 AND JOB = 'MANAGER');
```

For this query:

- EMP is a table with columns EMPNO, EMPNAME, DEPT, JOB, AGE, and SAL.
- EMPX1 is an index on EMP with key column AGE.
- EMPX2 is an index on EMP with key column JOB.

The plan table contains a sequence of rows describing the access. For this query, ACCESSTYPE uses the following values:

Value	Meaning
M	Start of multiple index access processing
MX	Indexes are to be scanned for later union or intersection
MI	An intersection (AND) is performed
MU	A union (OR) is performed

The following steps relate to the previous query and the values shown for the plan table in Figure 197 on page 688:

1. Index EMPX1, with matching predicate AGE= 34, provides a set of candidates for the result of the query. The value of MIXOPSEQ is 1.

2. Index EMPX1, with matching predicate AGE = 40, also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 2.

3. Index EMPX2, with matching predicate JOB='MANAGER', also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 3.

4. The first intersection (AND) is done, and the value of MIXOPSEQ is 4. This MI removes the two previous candidate lists (produced by MIXOPSEQs 2 and 3) by intersecting them to form an intermediate candidate list, IR1, which is not shown in PLAN_TABLE.

5. The last step, where the value MIXOPSEQ is 5, is a union (OR) of the two remaining candidate lists, which are IR1 and the candidate list produced by MIXOPSEQ 1. This final union gives the result for the query.

PLAN-NO	TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	PREFETCH	MIXOP-SEQ
1	EMP	M	0		L	0
1	EMP	MX	1	EMPX1		1
1	EMP	MX	1	EMPX1		2
1	EMP	MI	0			3
1	EMP	MX	1	EMPX2		4
1	EMP	MU	0			5

Figure 197. Plan table output for a query that uses multiple indexes. Depending on the filter factors of the predicates, the access steps can appear in a different order.

In this example, the steps in the multiple index access follow the physical sequence of the predicates in the query. This is not always the case. The multiple index steps are arranged in an order that uses RID pool storage most efficiently and for the least amount of time.

One-fetch access (ACCESSTYPE=I1)

One-fetch index access requires retrieving only one row. It is the best possible access path and is chosen whenever it is available. It applies to a statement with a MIN or MAX column function: the order of the index allows a single row to give the result of the function.

One-fetch index access is a possible access path when:

- There is only one table in the query.
- There is only one column function (either MIN or MAX).
- Either no predicate or all predicates are matching predicates for the index.
- There is no GROUP BY.
- Column functions are on:
  - The first index column if there are no predicates
  - The last matching column of the index if the last matching predicate is a range type
  - The next index column (after the last matching column) if all matching predicates are equal type

Queries using one-fetch index access: Assuming that an index exists on T(C1,C2,C3), the following queries use one-fetch index scan:

```
SELECT MIN(C1) FROM T;
SELECT MIN(C1) FROM T WHERE C1>5;
SELECT MIN(C1) FROM T WHERE C1>5 AND C1<10;
SELECT MIN(C2) FROM T WHERE C1=5;
SELECT MAX(C1) FROM T;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2<10;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2>5 AND C2<10;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2 BETWEEN 5 AND 10;
```

Index-only access (INDEXONLY=Y)

With *index-only access*, the access path does not require any data pages because the access information is available in the index. Conversely, when an SQL statement requests a column that is not in the index, updates any column in the table, or deletes a row, DB2 has to access the associated data pages. Because the index is almost always smaller than the table itself, an index-only access path usually processes the data efficiently.

With an index on T(C1,C2), the following queries can use index-only access:

```
SELECT C1, C2 FROM T WHERE C1 > 0;
SELECT C1, C2 FROM T;
SELECT COUNT(*) FROM T WHERE C1 = 1;
```

Equal unique index (MATCHCOLS=number of index columns)

An index that is fully matched and unique, and in which all matching predicates are equal-predicates, is called an *equal unique index* case. This case guarantees that only one row is retrieved. If there is no one-fetch index access available, this is considered the most efficient access over all other indexes that are not equal unique. (The uniqueness of an index is determined by whether or not it was defined as unique.)

Sometimes DB2 can determine that an index that is not fully matching is actually an equal unique index case. Assume the following case:

```
Unique Index1: (C1, C2)
Unique Index2: (C2, C1, C3)

SELECT C3 FROM T
 WHERE C1 = 1 AND
       C2 = 5;
```

Index1 is a fully matching equal unique index. However, Index2 is also an equal unique index even though it is not fully matching. Index2 is the better choice because, in addition to being equal and unique, it also provides index-only access.

UPDATE using an index

If no index key columns are updated, you can use an index while performing an UPDATE operation.

To use a matching index scan to update an index in which its key columns are being updated, the following conditions must be met:

- Each updated key column must have a corresponding predicate of the form "index_key_column = constant" or "index_key_column IS NULL".
- If a view is involved, WITH CHECK OPTION must not be specified.

With list prefetch or multiple index access, any index or indexes can be used in an UPDATE operation. Of course, to be chosen, those access paths must provide efficient access to the data

Interpreting access to two or more tables (join)

A *join* operation retrieves rows from more than one table and combines them. The operation specifies at least two tables, but they need not be distinct.

This section begins with "Definitions and examples" on page 690, below, and continues with descriptions of the methods of joining that can be indicated in a plan table:
- "Nested loop join (METHOD=1)" on page 692
- "Merge scan join (METHOD=2)" on page 693
- "Hybrid join (METHOD=4)" on page 695
- "Star schema (star join)" on page 697

Definitions and examples

Composite **TJ** → (Method 1) Nested loop join ← **TK** New

↓ Composite **Work File** → (Method 2) Merge scan join ← **TL** New

(Sort) ↓ Result
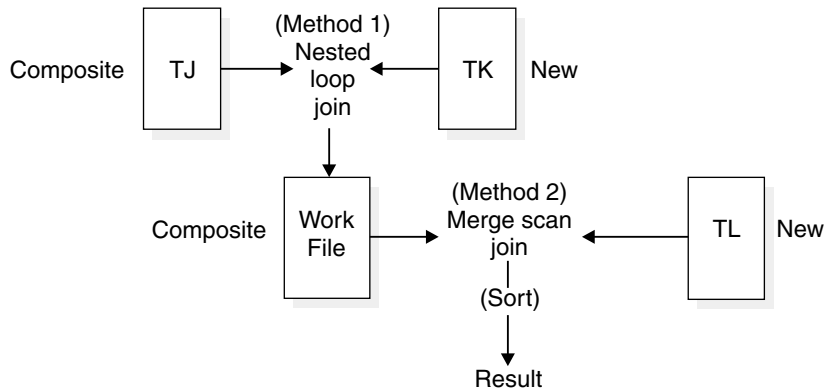
METHOD	TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	TSLOCK-MODE
0	TJ	I	1	TJX1	N	IS
1	TK	I	1	TKX1	N	IS
2	TL	I	0	TLX1	Y	S
3			0		N	

SORTN UNIQ	SORTN JOIN	SORTN ORDERBY	SORTN GROUPBY	SORTC UNIQ	SORTC JOIN	SORTC ORDERBY	SORTC GROUPBY
N	N	N	N	N	N	N	N
N	N	N	N	N	N	N	N
N	Y	N	N	N	Y	N	N
N	N	N	N	N	N	Y	N

Figure 198. Join methods as displayed in a plan table

A join operation can involve more than two tables. But the operation is carried out in a series of steps. Each step joins only two tables.

Definitions: The *composite table* (or *outer table*) in a join operation is the table remaining from the previous step, or it is the first table accessed in the first step. (In the first step, then, the composite table is composed of only one table.) The *new table* (or *inner table*) in a join operation is the table newly accessed in the step.

Example: Figure 198 shows a subset of columns in a plan table. In four steps, DB2:

1. Accesses the first table (METHOD=0), named TJ (TNAME), which becomes the composite table in step 2.
2. Joins the new table TK to TJ, forming a new composite table.
3. Sorts the new table TL (SORTN_JOIN=Y) and the composite table (SORTC_JOIN=Y), and then joins the two sorted tables.
4. Sorts the final composite table (TNAME is blank) into the desired order (SORTC_ORDERBY=Y).

Definitions: A join operation typically matches a row of one table with a row of another on the basis of a *join condition*. For example, the condition might specify that the value in column A of one table equals the value of column X in the other table (`WHERE T1.A = T2.X`).

Two kinds of joins differ in what they do with rows in one table that do not match on the join condition with any row in the other table:

- An *inner join* discards rows of either table that do not match any row of the other table.
- An *outer join* keeps unmatched rows of one or the other table, or of both. A row in the composite table that results from an unmatched row is filled out with null values. Outer joins are distinguished by which unmatched rows they keep.

Table 76. Join types and kept unmatched rows

This outer join:	Keeps unmatched rows from:
Left outer join	The composite (outer) table
Right outer join	The new (inner) table
Full outer join	Both tables

Example: Figure 199 shows an outer join with a subset of the values it produces in a plan table for the applicable rows. Column JOIN_TYPE identifies the type of outer join with one of these values:
- F for FULL OUTER JOIN
- L for LEFT OUTER JOIN
- Blank for INNER JOIN or no join

At execution, DB2 converts every right outer join to a left outer join; thus JOIN_TYPE never identifies a right outer join specifically.

```
EXPLAIN PLAN SET QUERYNO = 10 FOR
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
        PRODUCT, PART, UNITS
   FROM PROJECTS LEFT JOIN
      (SELECT PART,
          COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
          PRODUCTS.PRODUCT
          FROM PARTS FULL OUTER JOIN PRODUCTS
              ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
   ON PROJECTS.PROD# = PRODNUM
```

QUERYNO	QBLOCKNO	PLANNO	TNAME	JOIN_TYPE
10	1	1	PROJECTS	
10	1	2	TEMP	L
10	2	1	PRODUCTS	
10	2	2	PARTS	F

Figure 199. Plan table output for an example with outer joins

Materialization with outer join: Sometimes DB2 has to materialize a result table when an outer join is used in conjunction with other joins, views, or nested table expressions. You can tell when this happens by looking at the TABLE_TYPE and TNAME columns of the plan table. When materialization occurs, TABLE_TYPE

Nested loop join (METHOD=1)

This section describes this common join method.

```
SELECT A, B, X, Y
FROM (SELECT FROM OUTERT WHERE A=10)
LEFT JOIN INNERT ON B=X;
```

Left outer join using nested loop join

Table ⟶ OUTERT   INNERT   Composite
Columns ⟶ A   B   X   Y   A B X Y

A	B		X	Y		A	B	X	Y
10	3		5	A		10	3	3	B
10	1		3	B		10	1	1	D
			2	C		10	2	2	C
			1	D		10	2	2	E
10	2		2	E		10	6	-	-
10	6		9	F		10	1	1	D
10	1		7	G					

Scan the outer table.   The nested loop join
For each qualifying row . . . . find all matching rows   produces this result,
                                        in the inner table, by a   preserving the values
                                        table space or index scan.   of the outer table.

Figure 200. Nested Loop Join for a Left Outer Join

Method of joining

DB2 scans the composite (outer) table. For each row in that table that qualifies (by
satisfying the predicates on that table), DB2 searches for matching rows of the new
(inner) table. It concatenates any it finds with the current row of the composite
table. If no rows match the current row, then:

  For an inner join, DB2 discards the current row.
  For an outer join, DB2 concatenates a row of null values.

Stage 1 and stage 2 predicates eliminate unqualified rows during the join. (For an
explanation of those types of predicate, see "Stage 1 and stage 2 predicates" on
page 626.) DB2 can scan either table using any of the available access methods,
including table space scan.

Performance considerations

The nested loop join repetitively scans the inner table. That is, DB2 scans the outer
table once, and scans the inner table as many times as the number of qualifying
rows in the outer table. Hence, the nested loop join is usually the most efficient join
method when the values of the join column passed to the inner table are in
sequence and the index on the join column of the inner table is clustered, or the
number of rows retrieved in the inner table through the index is small.

When it is used

Nested loop join is often used if:

- The outer table is small.

- Predicates with small filter factors reduce the number of qualifying rows in the
  outer table.

- An efficient, highly clustered index exists on the join columns of the inner table.
- The number of data pages accessed in the inner table is small.

Example: left outer join: Figure 200 on page 692 illustrates a nested loop for a left outer join. The outer join preserves the unmatched row in OUTERT with values A=10 and B=6. The same join method for an inner join differs only in discarding that row.

Example: one-row table priority: For a case like the example below, with a unique index on T1.C2, DB2 detects that T1 has only one row that satisfies the search condition. DB2 makes T1 the first table in a nested loop join.

```
SELECT * FROM T1, T2
   WHERE  T1.C1 = T2.C1 AND
          T1.C2 = 5;
```

Example: Cartesian join with small tables first: A *Cartesian join* is a form of nested loop join in which there are no join predicates between the two tables. DB2 usually avoids a Cartesian join, but sometimes it is the most efficient method, as in the example below. The query uses three tables: T1 has 2 rows, T2 has 3 rows, and T3 has 10 million rows.

```
SELECT * FROM T1, T2, T3
   WHERE  T1.C1 = T3.C1 AND
          T2.C2 = T3.C2 AND
          T3.C3 = 5;
```

Join predicates are between T1 and T3 and between T2 and T3. There is no join predicate between T1 and T2.

Assume that 5 million rows of T3 have the value C3=5. Processing time is large if T3 is the outer table of the join and tables T1 and T2 are accessed for each of 5 million rows.

But if all rows from T1 and T2 are joined, without a join predicate, the 5 million rows are accessed only six times, once for each row in the Cartesian join of T1 and T2. It is difficult to say which access path is the most efficient. DB2 evaluates the different options and could decide to access the tables in the sequence T1, T2, T3.

Sorting the composite table: Your plan table could show a nested loop join that includes a sort on the composite table. DB2 might sort the composite table (the outer table in Figure 200) if the following conditions exist:
- The join columns in the composite table and the new table are not in the same sequence.
- The join column of the composite table has no index.
- The index is poorly clustered.

Nested loop join with a sorted composite table uses sequential detection efficiently to prefetch data pages of the new table, reducing the number of synchronous I/O operations and the elapsed time.

Merge scan join (METHOD=2)

Merge scan join is also known as *merge join* or *sort merge join*. For this method, there must be one or more predicates of the form `TABLE1.COL1=TABLE2.COL2`, where the two columns have the same data type and length attribute.

Method of joining

Figure 201 illustrates a merge scan join.

```
SELECT A, B, X, Y
FROM OUTER, INNER
WHERE A=10 AND B=X;
```

Merge scan join

Condense and sort the outer table, or access it through an index on column B.

Condense and sort the inner table.

Table ⟶
Columns ⟶

OUTER			INNER			Composite			
A	B		X	Y		A	B	X	Y
10	1		1	D		10	1	1	D
10	1		2	C		10	1	1	D
10	2		2	E		10	2	2	C
10	3		3	B		10	2	2	E
10	6		5	A		10	3	3	B
			7	G					
			9	F					

Scan the outer table.
For each row, ... scan a group of matching rows in the inner table.

The merge scan join produces this result.

Figure 201. Merge scan join

DB2 scans both tables in the order of the join columns. If no efficient indexes on the join columns provide the order, DB2 might sort the outer table, the inner table, or both. The inner table is put into a work file; the outer table is put into a work file only if it must be sorted. When a row of the outer table matches a row of the inner table, DB2 returns the combined rows.

DB2 then reads another row of the inner table that might match the same row of the outer table and continues reading rows of the inner table as long as there is a match. When there is no longer a match, DB2 reads another row of the outer table.

- If that row has the same value in the join column, DB2 reads again the matching group of records from the inner table. Thus, a group of duplicate records in the inner table is scanned as many times as there are matching records in the outer table.
- If the outer row has a new value in the join column, DB2 searches ahead in the inner table. It can find any of the following rows:
  - Unmatched rows in the inner table, with lower values in the join column.
  - A new matching inner row. DB2 then starts the process again.
  - An inner row with a higher value of the join column. Now the row of the outer table is unmatched. DB2 searches ahead in the outer table, and can find any of the following rows:
    - Unmatched rows in the outer table.
    - A new matching outer row. DB2 then starts the process again.
    - An outer row with a higher value of the join column. Now the row of the inner table is unmatched, and DB2 resumes searching the inner table.

If DB2 finds an unmatched row:

For an inner join, DB2 discards the row.

For a left outer join, DB2 discards the row if it comes from the inner table and keeps it if it comes from the outer table.

For a full outer join, DB2 keeps the row.

When DB2 keeps an unmatched row from a table, it concatenates a set of null values as if that matched from the other table. A merge scan join must be used for a full outer join.

Performance considerations

A full outer join by this method uses all predicates in the ON clause to match the two tables and reads every row at the time of the join. Inner and left outer joins use only stage 1 predicates in the ON clause to match the tables. If your tables match on more than one column, it is generally more efficient to put all the predicates for the matches in the ON clause, rather than to leave some of them in the WHERE clause.

For an inner join, DB2 can derive extra predicates for the inner table at bind time and apply them to the sorted outer table to be used at run time. The predicates can reduce the size of the work file needed for the inner table.

If DB2 has used an efficient index on the join columns, to retrieve the rows of the inner table, those rows are already in sequence. DB2 puts the data directly into the work file without sorting the inner table, which reduces the elapsed time.

When it is used

A merge scan join is often used if:

- The qualifying rows of the inner and outer table are large, and the join predicate does not provide much filtering; that is, in a many-to-many join.
- The tables are large and have no indexes with matching columns.
- Few columns are selected on inner tables. This is the case when a DB2 sort is used. The fewer the columns to be sorted, the more efficient the sort is.

Hybrid join (METHOD=4)

The method applies only to an inner join and requires an index on the join column of the inner table.
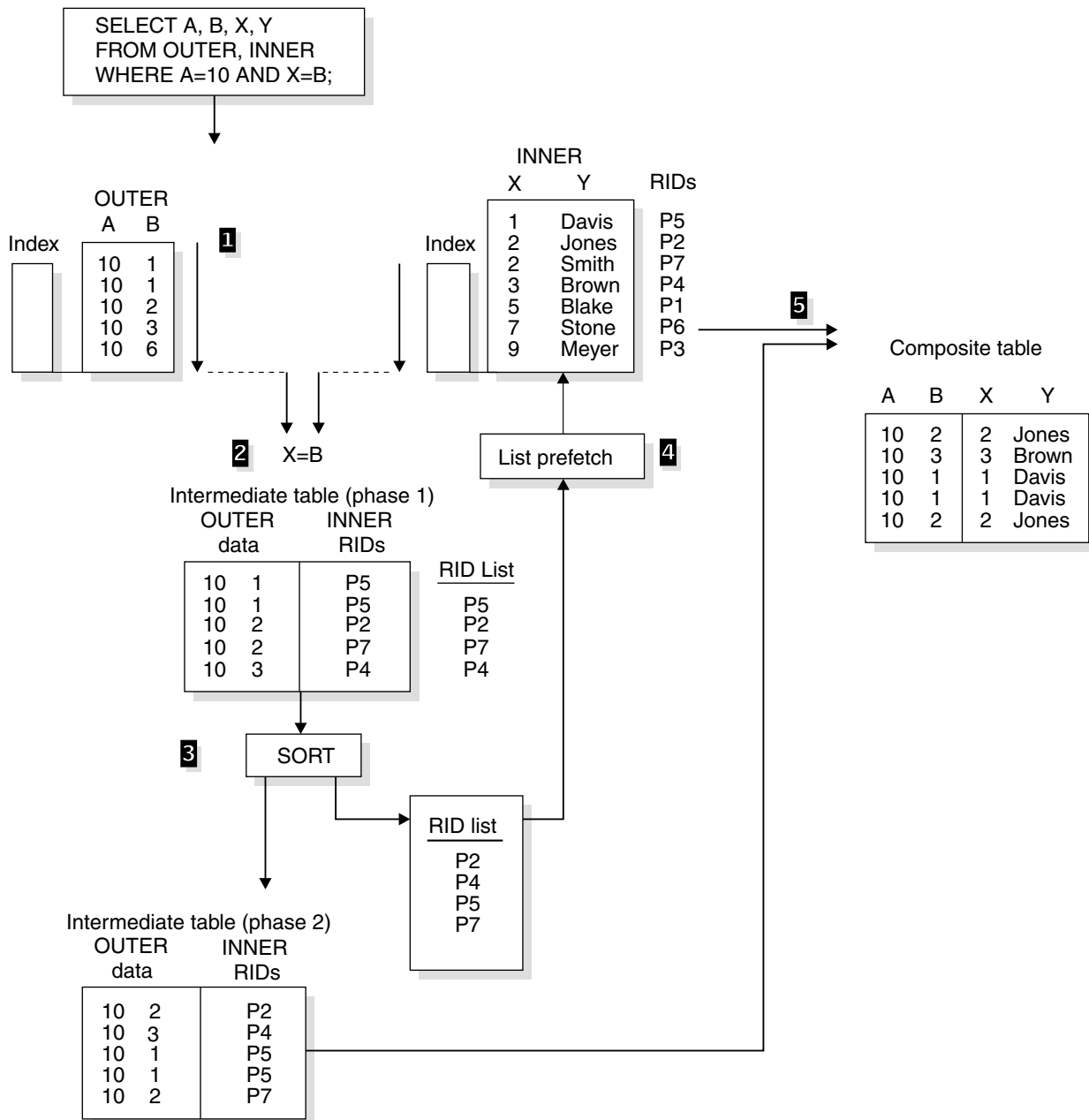
Figure 202. Hybrid join (SORTN_JOIN='Y')

Method of joining

The method requires obtaining RIDs in the order needed to use list prefetch. The steps are shown in Figure 202. In that example, both the outer table (OUTER) and the inner table (INNER) have indexes on the join columns.

In the successive steps, DB2:

1 Scans the outer table (OUTER).

2 Joins the outer tables with RIDs from the index on the inner table. The result is the phase 1 intermediate table. The index of the inner table is scanned for every row of the outer table.

3 Sorts the data in the outer table and the RIDs, creating a sorted RID list and the phase 2 intermediate table. The sort is indicated by a value of Y in column SORTN_JOIN of the plan table. If the index on the inner table is a clustering index, DB2 can skip this sort; the value in SORTN_JOIN is then N.

4 Retrieves the data from the inner table, using list prefetch.

5 Concatenates the data from the inner table and the phase 2 intermediate table to create the final composite table.

Possible results from EXPLAIN for hybrid join

Column Value	Explanation
METHOD='4'	A hybrid join was used.
SORTC_JOIN='Y'	The composite table was sorted.
SORTN_JOIN='Y'	The intermediate table was sorted in the order of inner table RIDs. A non-clustered index accessed the inner table RIDs.
SORTN_JOIN='N'	The intermediate table RIDs were not sorted. A clustered index retrieved the inner table RIDs, and the RIDs were already well ordered.
PREFETCH='L'	Pages were read using list prefetch.

Performance considerations

Hybrid join uses list prefetch more efficiently than nested loop join, especially if there are indexes on the join predicate with low cluster ratios. It also processes duplicates more efficiently because the inner table is scanned only once for each set of duplicate values in the join column of the outer table.

If the index on the inner table is highly clustered, there is no need to sort the intermediate table (SORTN_JOIN=N). The intermediate table is placed in a table in memory rather than in a work file.

When it is used

Hybrid join is often used if:

- A nonclustered index or indexes are used on the join columns of the inner table.
- The outer table has duplicate qualifying rows.

Star schema (star join)

A star schema or star join is a logical database design that is included in decision support applications. A star schema is composed of a fact table and a number of dimension tables that are connected to it. A dimension table contains several values that are given an ID, which is used in the fact table instead of all the values.

You can think of the fact table, which is much larger than the dimension tables, as being in the center surrounded by dimension tables; the result resembles a star formation. The following diagram illustrates the star formation:
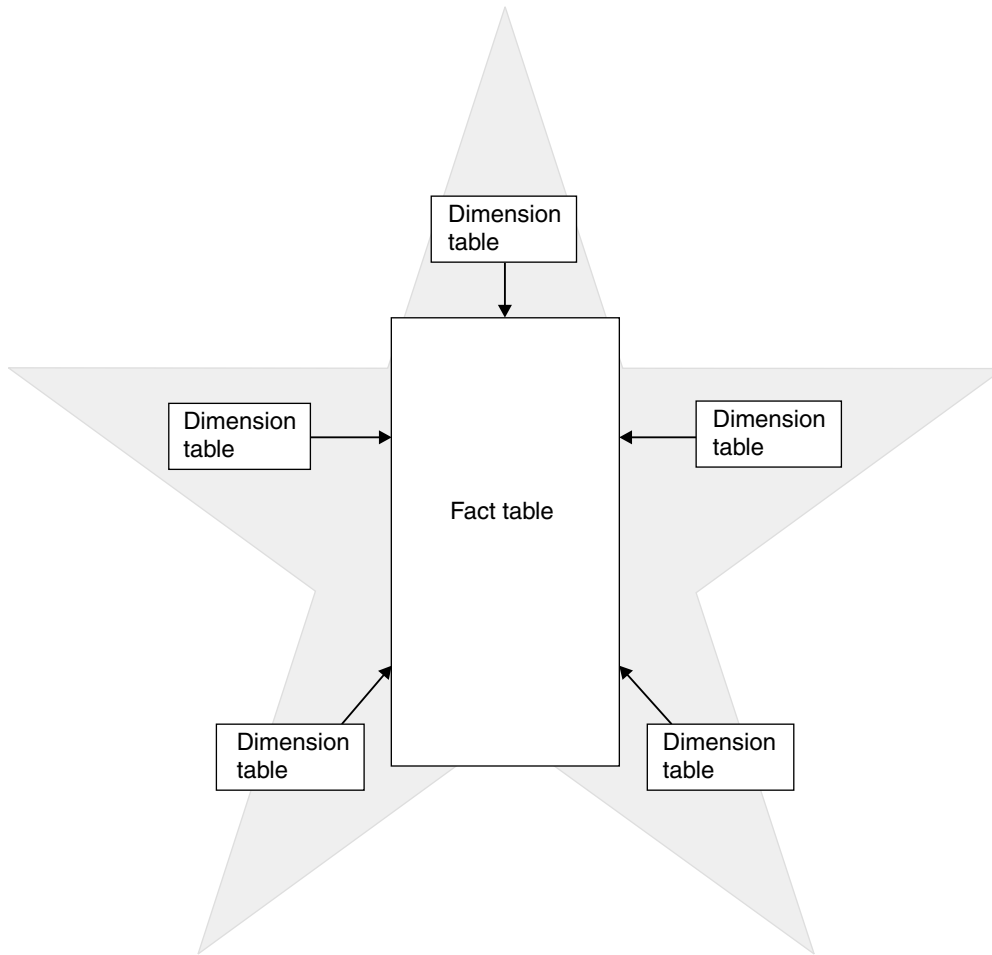
Figure 203. Star schema with a fact table and dimension tables

Example
For an example of a star schema, consider the following scenario. A star schema is composed of a fact table for sales, with dimension tables connected to it for time, products, and geographic locations. The time table has an ID for each month, its quarter, and the year. The product table has an ID for each product item and its class and its inventory. The geographic location table has an ID for each city and its country.

In this scenario, the sales table contains three columns with IDs from the dimension tables for time, product, and location instead of three columns for time, three columns for products, and two columns for location. Thus, the size of the fact table is greatly reduced. In addition, if you needed to change an item, you would do it once in a dimension table instead of several times for each instance of the item in the fact table.

You can create even more complex star schemas by breaking a dimension table into a fact table with its own dimension tables. The fact table would be connected to the main fact table.

When it is used

To access the data in a star schema, you write SELECT statements that include join operations between the fact table and the dimension tables; no join operations exist between dimension tables. When the query meets the following conditions, that query is a star schema:

- The query references at least two dimensions.
- All join predicates are between the fact table and the dimension tables, or within tables of the same dimension.
- All join predicates between the fact table and dimension tables are equi-join predicates.
- All join predicates between the fact table and dimension tables are Boolean term predicates. For more information, see "Boolean term (BT) predicates" on page 626.
- No correlated subqueries cross dimensions.
- No single fact table column is joined to columns of different dimension tables in join predicates. For example, fact table column F1 cannot be joined to column D1 of dimension table T1 and also joined to column D2 of dimension table T2.
- After DB2 simplifies join operations, no outer join operations exist. For more information, see "When DB2 simplifies join operations" on page 638.
- The data type and length of both sides of a join predicate are the same.
- The value of subsystem parameter STARJOIN is 1, or the cardinality of the fact table to the largest dimension table meets the requirements specified by the value of the subsystem parameter. The values of STARJOIN and cardinality requirements are:

  -1   Star join is disabled. This is the default.

  1   Star join is enabled. The one table with the largest cardinality is the fact table. However, if there is more than one table with this cardinality, star join is not enabled.

  0   Star join is enabled if the cardinality of the fact table is at least 25 times the cardinality of the largest dimension that is a base table that is joined to the fact table.

  n   Star join is enabled if the cardinality of the fact table is at least *n* times the cardinality of the largest dimension that is a base table that is joined to the fact table, where $2 \leq n \leq 32768$.

Examples: query with three dimension tables: Suppose you have a store in San Jose and want information about sales of audio equipment from that store in 2000. For this example, you want to join the following tables:

- A fact table for SALES (S)
- A dimension table for TIME (T) with columns for an ID, month, quarter, and year
- A dimension table for geographic LOCATION (L) with columns for an ID, city, region, and country
- A dimension table for PRODUCT (P) with columns for an ID, product item, class, and inventory

You could write the following query to join the tables:

```
SELECT *
FROM SALES S, TIME T, PRODUCT P, LOCATION L
WHERE S.TIME = T.ID   AND
S.PRODUCT = P.ID AND
S.LOCATION = L.ID  AND
T.YEAR = 2000 AND
P.CLASS = 'SAN JOSE';
```

You would use the following index:

```
CREATE INDEX  XSALES_TPL ON SALES (TIME, PRODUCT, LOCATION);
```

Your EXPLAIN output looks like the following table;

QUERYNO	QUERY-BLOCKNO	METHOD	TNAME	JOIN TYPE	SORTN JOIN
1	1	0	TIME	S	
1	1	1	PRODUCT	S	
1	1	1	LOCATION	S	
1	1	1	SALES	S	

Figure 204. Plan table output for a star join example with TIME, PRODUCT, and LOCATION

For another example, suppose you want to use the same SALES (S), TIME (T), PRODUCT (P), and LOCATION (L) tables for a similar query and index; however, for this example the index does not include the TIME dimension. A query doesn't have to involve all dimensions. In this example, the star join is performed on one query block at stage 1 and a star join is performed on another query block at stage 2.

You could write the following query to join the tables:

```
SELECT *
FROM SALES S, TIME T, PRODUCT P, LOCATION L
WHERE S.TIME = T.ID  AND
S.PRODUCT = P.ID AND
S.LOCATION = L.ID  AND
T.YEAR = 2000 AND
P.CLASS = 'AUDIO';
```

You would use the following index:

```
CREATE INDEX  XSALES_TPL ON SALES (PRODUCT, LOCATION);
```

Your EXPLAIN output looks like the following table;

QUERYNO	QUERY-BLOCKNO	METHOD	TNAME	JOIN TYPE	SORTN JOIN
1	1	0	TIME	S	
1	1	2	DSNWFQB(02) (Note 1)	S	Y
1	2	0	PRODUCT	S (Note 2)	
1	2	1	LOCATION	S (Note 2)	
1	2	1	SALES	S (Note 2)	

Notes to Figure 205:

1. This star join is handled at stage 2; the tables in this query block are joined with a merge scan join (METHOD = 2).
2. This star join is handled at stage 1; the tables in this query block are joined with a nested loop join (METHOD = 1).

Figure 205. Plan table output for a star join example with PRODUCT and LOCATION

Interpreting data prefetch

Prefetch is a mechanism for reading a set of pages, usually 32, into the buffer pool with only one asynchronous I/O operation. Prefetch can allow substantial savings in both processor cycles and I/O costs. To achieve those savings, monitor the use of prefetch.

A plan table can indicate the use of two kinds of prefetch:

- "Sequential prefetch (PREFETCH=S)"
- "List prefetch (PREFETCH=L)" on page 702

If DB2 does not choose prefetch at bind time, it can sometimes use it at execution time nevertheless. The method is described in "Sequential detection at execution time" on page 703.

Sequential prefetch (PREFETCH=S)

Sequential prefetch reads a sequential set of pages. The maximum number of pages read by a request issued from your application program is determined by the size of the buffer pool used. For each buffer pool size (4 KB, 8 KB, 16 KB, and 32 KB), Table 77 on page 702 shows the number pages read by prefetch for each asynchronous I/O.

Table 77. The number of pages read by prefetch, by buffer pool size

Buffer pool size	Number of buffers	Pages read by prefetch (for each asynchronous I/O)
4 KB	<=223 buffers	8 pages
	224-999 buffers	16 pages
	1000+ buffers	32 pages
8 KB	<=112 buffers	4 pages
	113-499 buffers	8 pages
	500+ buffers	16 pages
16 KB	<=56 buffers	2 pages
	57-249 buffers	4 pages
	250+ buffers	8 pages
32 KB	<=16 buffers	0 pages (prefetch disabled)
	17-99 buffers	2 pages
	100+ buffers	4 pages

For certain utilities (LOAD, REORG, RECOVER), the prefetch quantity can be twice as much.

When it is used: Sequential prefetch is generally used for a table space scan.

For an index scan that accesses 8 or more consecutive data pages, DB2 requests sequential prefetch at bind time. The index must have a cluster ratio of 80% or higher. Both data pages and index pages are prefetched.

List prefetch (PREFETCH=L)

List prefetch reads a set of data pages determined by a list of RIDs taken from an index. The data pages need not be contiguous. The maximum number of pages that can be retrieved in a single list prefetch is 32 (64 for utilities).

List prefetch can be used in conjunction with either single or multiple index access.

The access method
List prefetch uses the following three steps:
1. RID retrieval: A list of RIDs for needed data pages is found by matching index scans of one or more indexes.
2. RID sort: The list of RIDs is sorted in ascending order by page number.
3. Data retrieval: The needed data pages are prefetched in order using the sorted RID list.

List prefetch does not preserve the data ordering given by the index. Because the RIDs are sorted in page number order before accessing the data, the data is not retrieved in order by any column. If the data must be ordered for an ORDER BY clause or any other reason, it requires an additional sort.

In a hybrid join, if the index is highly clustered, the page numbers might not be sorted before accessing the data.

List prefetch can be used with most matching predicates for an index scan. IN-list predicates are the exception; they cannot be the matching predicates when list prefetch is used.

When it is used

List prefetch is used:

- Usually with a single index that has a cluster ratio lower than 80%
- Sometimes on indexes with a high cluster ratio, if the estimated amount of data to be accessed is too small to make sequential prefetch efficient, but large enough to require more than one regular read
- Always to access data by multiple index access
- Always to access data from the inner table during a hybrid join

Bind time and execution time thresholds

DB2 does not consider list prefetch if the estimated number of RIDs to be processed would take more than 50% of the RID pool when the query is executed. You can change the size of the RID pool in the field RID POOL SIZE on installation panel DSNTIPC. The maximum size of a RID pool is 1000MB. The maximum size of a single RID list is approximately 16 million RIDs. For information on calculating RID pool size, see Part 5 (Volume 2) of *DB2 Administration Guide*.

During execution, DB2 ends list prefetching if more than 25% of the rows in the table (with a minimum of 4075) must be accessed. Record IFCID 0125 in the performance trace, mapped by macro DSNDQW01, indicates whether list prefetch ended.

When list prefetch ends, the query continues processing by a method that depends on the current access path.

- For access through a single index or through the union of RID lists from two indexes, processing continues by a table space scan.
- For index access before forming an intersection of RID lists, processing continues with the next step of multiple index access. If no step remains and no RID list has been accumulated, processing continues by a table space scan.

While forming an intersection of RID lists, if any list has 32 or fewer RIDs, intersection stops and the list of 32 or fewer RIDs is used to access the data.

Sequential detection at execution time

If DB2 does not choose prefetch at bind time, it can sometimes use it at execution time nevertheless. The method is called *sequential detection*.

When it is used

DB2 can use sequential detection for both index leaf pages and data pages. It is most commonly used on the inner table of a nested loop join, if the data is accessed sequentially.

If a table is accessed repeatedly using the same statement (for example, DELETE in a do-while loop), the data or index leaf pages of the table can be accessed sequentially. This is common in a batch processing environment. Sequential detection can then be used if access is through:

- SELECT or FETCH statements
- UPDATE and DELETE statements
- INSERT statements when existing data pages are accessed sequentially

DB2 can use sequential detection if it did not choose sequential prefetch at bind time because of an inaccurate estimate of the number of pages to be accessed.

Sequential detection is not used for an SQL statement that is subject to referential constraints.

How to tell whether it was used

A plan table does not indicate sequential detection, which is not determined until run time. You can determine whether sequential detection was used from record IFCID 0003 in the accounting trace or record IFCID 0006 in the performance trace.

How to tell if it might be used

The pattern of accessing a page is tracked when the application scans DB2 data through an index. Tracking is done to detect situations where the access pattern that develops is sequential or nearly sequential.

The most recent 8 pages are tracked. A page is considered page-sequential if it is within P/2 advancing pages of the current page, where P is the prefetch quantity. P is usually 32.

If a page is page-sequential, DB2 determines further if data access is sequential or nearly sequential. Data access is declared sequential if more than 4 out of the last 8 pages are page-sequential; this is also true for index-only access. The tracking is continuous, allowing access to slip into and out of data access sequential.

When data access sequential is first declared, which is called *initial data access* sequential, three page ranges are calculated as follows:

- Let A be the page being requested. RUN1 is defined as the page range of length P/2 pages starting at A.
- Let B be page A + P/2. RUN2 is defined as the page range of length P/2 pages starting at B.
- Let C be page B + P/2. RUN3 is defined as the page range of length P pages starting at C.

For example, assume page A is 10, the following figure illustrates the page ranges that DB2 calculates.

	A	B	C	
	RUN1	RUN2	RUN3	
Page #	10	26	42	
P=32 pages	16	16	32	
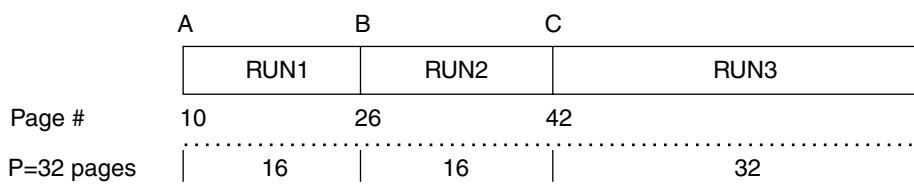
Figure 206. Initial page ranges to determine when to prefetch

For initial data access sequential, prefetch is requested starting at page A for P pages (RUN1 and RUN2). The prefetch quantity is always P pages.

For subsequent page requests where the page is 1) page sequential and 2) data access sequential is still in effect, prefetch is requested as follows:

- If the desired page is in RUN1, then no prefetch is triggered because it was already triggered when data access sequential was first declared.
- If the desired page is in RUN2, then prefetch for RUN3 is triggered and RUN2 becomes RUN1, RUN3 becomes RUN2, and RUN3 becomes the page range starting at C+P for a length of P pages.

If a data access pattern develops such that data access sequential is no longer in effect and, thereafter, a new pattern develops that is sequential as described above, then initial data access sequential is declared again and handled accordingly.

Because, at bind time, the number of pages to be accessed can only be estimated, sequential detection acts as a safety net and is employed when the data is being accessed sequentially.

In extreme situations, when certain buffer pool thresholds are reached, sequential prefetch can be disabled. For a description of buffer pools and thresholds, see Part 5 (Volume 2) of *DB2 Administration Guide* .

Determining sort activity

DB2 can use two general types of sorts that DB2 can use when accessing data. One is a sort of data rows; the other is a sort of row identifiers (RIDs) in a RID list.

Sorts of data

After you run EXPLAIN, DB2 sorts are indicated in PLAN_TABLE. The sorts can be either sorts of the composite table or the new table. If a single row of PLAN_TABLE has a 'Y' in more than one of the sort composite columns, then one sort accomplishes two things. (DB2 will not perform two sorts when two 'Y's are in the same row.) For instance, if both SORTC_ORDERBY and SORTC_UNIQ are 'Y' in one row of PLAN_TABLE, then a single sort puts the rows in order and removes any duplicate rows as well.

The only reason DB2 sorts the new table is for join processing, which is indicated by SORTN_JOIN.

Sorts for group by and order by
These sorts are indicated by SORTC_ORDERBY, and SORTC_GROUPBY in PLAN_TABLE. If there is both a GROUP BY clause and an ORDER BY clause, and if every item in the ORDER-BY list is in the GROUP-BY list, then only one sort is performed, which is marked as SORTC_ORDERBY.

The performance of the sort by the GROUP BY clause is improved when the query accesses a single table and when the GROUP BY column has no index.

Sorts to remove duplicates
This type of sort is used to process a query with SELECT DISTINCT, with a set function such as COUNT(DISTINCT COL1), or to remove duplicates in UNION processing. It is indicated by SORTC_UNIQ in PLAN_TABLE.

Sorts used in join processing
Before joining two tables, it is often necessary to first sort either one or both of them. For hybrid join (METHOD 4) and nested loop join (METHOD 1), the composite table can be sorted to make the join more efficient. For merge join (METHOD 2), both the composite table and new table need to be sorted unless an index is used for accessing these tables that gives the correct order already. The sorts needed for join processing are indicated by SORTN_JOIN and SORTC_JOIN in the PLAN_TABLE.

Sorts needed for subquery processing
When a noncorrelated IN or NOT IN subquery is present in the query, the results of the subquery are sorted and put into a work file for later reference by the parent query. The results of the subquery are sorted because this allows the parent query

to be more efficient when processing the IN or NOT IN predicate. Duplicates are not needed in the work file, and are removed. Noncorrelated subqueries used with =ANY or =ALL, or NOT=ANY or NOT=ALL also need the same type of sort as IN or NOT IN subqueries. When a sort for a noncorrelated subquery is performed, you see both SORTC_ORDERBY and SORTC_UNIQUE in PLAN_TABLE. This is because DB2 removes the duplicates and performs the sort.

SORTN_GROUPBY, SORTN_ORDERBY, and SORTN_UNIQ are not currently used by DB2.

Sorts of RIDs

To perform list prefetch, DB2 sorts RIDs into ascending page number order. This sort is very fast and is done totally in memory. A RID sort is usually not indicated in the PLAN_TABLE, but a RID sort normally is performed whenever list prefetch is used. The only exception to this rule is when a hybrid join is performed and a single, highly clustered index is used on the inner table. In this case SORTN_JOIN is 'N', indicating that the RID list for the inner table was not sorted.

The effect of sorts on OPEN CURSOR

The type of sort processing required by the cursor affects the amount of time it can take for DB2 to process the OPEN CURSOR statement. This section outlines the effect of sorts and parallelism on OPEN CURSOR.

Without parallelism:
- If no sorts are required, then OPEN CURSOR does not access any data. It is at the first fetch that data is returned.
- If a sort is required, then the OPEN CURSOR causes the materialized result table to be produced. Control returns to the application after the result table is materialized. If a cursor that requires a sort is closed and reopened, the sort is performed again.
- If there is a RID sort, but no data sort, then it is not until the first row is fetched that the RID list is built from the index and the first data record is returned. Subsequent fetches access the RID pool to access the next data record.

With parallelism:
- At OPEN CURSOR, parallelism is asynchronously started, regardless of whether a sort is required. Control returns to the application immediately after the parallelism work is started.
- If there is a RID sort, but no data sort, then parallelism is not started until the first fetch. This works the same way as with no parallelism.

Processing for views and nested table expressions

This section describes how DB2 processes views and nested table expressions. A nested table expression (which is called *table expression* in this description) is the specification of a subquery in the FROM clause of an SQL SELECT statement. The processing of table expressions is similar to a view. Two methods are used to satisfy your queries that reference views or table expressions:
- *Merge*
- *Materialization*

You can determine the methods that are used by executing EXPLAIN for the statement that contains the view or nested table expression. In addition, you can

use EXPLAIN to determine when UNION operators are used and how DB2 might
eliminate unnecessary subselects to improve the performance of a query.

Merge

The merge process is more efficient than materialization, as described in
"Performance of merge versus materialization" on page 711. In the merge process,
the statement that references the view or table expression is combined with the
fullselect that defined the view or table expression. This combination creates a
logically equivalent statement. This equivalent statement is executed against the
database.

Consider the following statements, one of which defines a view, the other of which
references the view:

```
View-defining statement:              View referencing statement:

CREATE VIEW VIEW1 (VC1,VC21,VC32) AS   SELECT VC1,VC21
SELECT C1,C2,C3 FROM T1                FROM VIEW1
WHERE C1 > C3;                         WHERE VC1 IN (A,B,C);
```

The fullselect of the view-defining statement can be merged with the
view-referencing statement to yield the following logically equivalent statement:

```
Merged statement:

SELECT C1,C2 FROM T1
WHERE C1 > C3 AND C1 IN (A,B,C);
```

Here is another example of when a view and table expression can be merged:

```
SELECT * FROM V1 X
  LEFT JOIN
    (SELECT * FROM T2) Y ON X.C1=Y.C1
                  LEFT JOIN T3 Z ON X.C1=Z.C1;
```

```
Merged statement:

SELECT * FROM V1 X
  LEFT JOIN
    T2 ON X.C1 = T2.C1
        LEFT JOIN T3 Z ON X.C1 = Z.C1;
```

Materialization

Views and table expressions cannot always be merged. Look at the following
statements:

```
View defining statement:               View referencing statement:

CREATE VIEW VIEW1 (VC1,VC2) AS         SELECT MAX(VC1)
SELECT SUM(C1),C2 FROM T1              FROM VIEW1;
GROUP BY C2;
```

Column VC1 occurs as the argument of a column function in the view referencing
statement. The values of VC1, as defined by the view-defining fullselect, are the
result of applying the column function SUM(C1) to groups after grouping the base
table T1 by column C2. No equivalent single SQL SELECT statement can be
executed against the base table T1 to achieve the intended result. There is no way
to specify that column functions should be applied successively.

Two steps of materialization
In the previous example, DB2 performs materialization of the view or table
expression, which is a two step process.

1. The fullselect that defines the view or table expression is executed against the database, and the results are placed in a temporary copy of a result table.
2. The statement that references the view or table expression is then executed against the temporary copy of the result table to obtain the intended result.

Whether materialization is needed depends upon the attributes of the referencing statement, or logically equivalent referencing statement from a prior merge, and the attributes of the fullselect that defines the view or table expression.

When views or table expressions are materialized

In general, DB2 uses materialization to satisfy a reference to a view or table expression when there is aggregate processing (grouping, column functions, distinct), indicated by the defining fullselect, in conjunction with either aggregate processing indicated by the statement referencing the view or table expression, or by the view or table expression participating in a join. For views and table expressions that are defined with UNION or UNION ALL, DB2 can often distribute aggregate processing, joins, and qualified predicates to avoid materialization. For more information, see "Using EXPLAIN to determine UNION activity and query rewrite" on page 709.

Table 78 indicates some cases in which materialization occurs. DB2 can also use materialization in statements that contain multiple outer joins, outer joins that combine with inner joins, or merges that cause a join of greater than 15 tables.

Table 78. Cases when DB2 performs view or table expression materialization. The "X" indicates a case of materialization. Notes follow the table.

A SELECT FROM a view or a table expression uses...(1)	View definition or table expression uses...(2)					
	GROUP BY	DISTINCT	Column function	Column function DISTINCT	UNION(4)	UNION ALL(4)
Joins (3)	X	X	X	X	X	-
GROUP BY	X	X	X	X	X	X
DISTINCT	-	X	-	X	X	X
Column function	X	X	X	X	X	X
Column function DISTINCT	X	X	X	X	X	X
SELECT subset of view or table expression columns	-	X	-	-	X	-

Notes to Table 78:

1. If the view is referenced as the target of an INSERT, UPDATE, or DELETE, then view merge is used to satisfy the view reference. Only updatable views can be the target in these statements. See Chapter 5 of *DB2 SQL Reference* for information on which views are read-only (not updatable).

   An SQL statement can reference a particular view multiple times where some of the references can be merged and some must be materialized.

2. If a SELECT list contains a host variable in a table expression, then materialization occurs. For example:

```
SELECT C1 FROM
    (SELECT :HV1 AS C1 FROM T1) X;
```

If a view or nested table expression is defined to contain a user-defined function, and if that user-defined function is defined as NOT DETERMINISTIC or EXTERNAL ACTION, then the view or nested table expression is always materialized.

3. Additional details about materialization with outer joins:

   - If a WHERE clause exists in a view or table expression, and it does not contain a column, materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT C1 FROM T1
         WHERE 1=1) X LEFT JOIN T2 Y
                        ON X.C1=Y.C1;
   ```

   - If the outer join is a full outer join and the SELECT list of the view or nested table expression does not contain a standalone column for the column that is used in the outer join ON clause, then materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT C1+10 AS C2 FROM T1) X FULL JOIN T2 Y
                        ON X.C2=Y.C2;
   ```

   - If there is no column in a SELECT list of a view or nested table expression, materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT 1+2+:HV1. AS C1 FROM T1) X LEFT JOIN T2 Y
                        ON X.C1=Y.C1;
   ```

4. Additional details about materialization with UNION or UNION ALL:

   - When the view is the operand in an outer join for which nulls are used for non-matching values, materialization occurs. This situation happens when the view is either operand in a full outer join, the right operand in a left outer join, or the left operand in a right outer join.

   - If the number of tables would exceed 255 after distribution, then distribution will not occur, and the result will be materialized.

Using EXPLAIN to determine when materialization occurs

For each reference to a view or table expression that is materialized, rows describing the access path for both steps of the materialization process appear in the PLAN_TABLE. These rows describe the access path used to formulate the temporary result indicated by the view's defining fullselect, and they describe the access to the temporary result as indicated by the referencing statement. The defining fullselect can also refer to views or table expressions that need to be materialized.

Another indication that DB2 chose view materialization is that the view name or table expression name appears as a TNAME attribute for rows describing the access path for the referencing query block. When DB2 chooses merge, EXPLAIN data for the merged statement appears in PLAN_TABLE; only the names of the base tables on which the view or table expression is defined appear.

Using EXPLAIN to determine UNION activity and query rewrite

For each reference to a view or table expression that is defined with UNION or UNION ALL operators, DB2 tries to rewrite the query into a logically equivalent statement with improved performance by:

- Distributing qualified predicates, joins, and aggregations across the subselects of UNION ALL. Such distribution helps to avoid materialization. No distribution is performed for UNION.

- Eliminating unnecessary subselects of the view or table expression. For DB2 to eliminate subselects, the referencing query and the view or table definition must have predicates that are based on common columns.

The QBLOCK_TYPE column in the plan table indicates union activity. For a UNION ALL, the column contains 'UNIONA'. For UNION, the column contains 'UNION'. When QBLOCK_TYPE='UNION', the METHOD column on the same row is set to 3 and the SORTC_UNIQ column is set to 'Y' to indicate that a sort is necessary to remove duplicates. As with other views and table expressions, the plan table also shows when DB2 uses materialization instead of merge.

Example: Consider the following statements, which define a view, reference the view, and show how DB2 rewrites the referencing statement. Figure 207 on page 711 shows a subset of columns in a plan table for the query. Notice how DB2 eliminates the second subselect of the view definition from the rewritten query and how the plan table indicates this removal by showing a UNION ALL for only the first and third subselect in the view definition.

```
View defining statement: View is created on three tables that contain weekly data

CREATE VIEW V1 (CUSTNO, CHARGES, DATE) as
  SELECT CUSTNO, CHARGES, DATE
  FROM WEEK1
  WHERE DATE BETWEEN '01/01/2000' And '01/07/2000'
UNION ALL
 SELECT CUSTNO, CHARGES, DATE
  FROM WEEK2
  WHERE DATE BETWEEN '01/08/2000' And '01/14/2000'
UNION ALL
  SELECT CUSTNO, CHARGES, DATE
  FROM WEEK3
  WHERE DATE BETWEEN '01/15/2000' And '01/21/2000';

View referencing statement: For each customer in California, find the average charges
during the first and third Friday of January 2000

SELECT V1.CUSTNO, AVG(V1.CHARGES)
  FROM CUST, V1
  WHERE CUST.CUSTNO=V1.CUSTNO
    AND CUST.STATE='CA'
    AND DATE IN ('01/07/2000','01/21/2000')
  GROUP BY V1.CUSTNO;

Rewritten statement (assuming that CHARGES is defined as NOT NULL):

SELECT CUSTNO_U, SUM(SUM_U)/SUM(CNT_U)
  FROM
  ( SELECT WEEK1.CUSTNO, SUM(CHARGES), COUNT(CHARGES)
      FROM CUST, WEEK1
      Where CUST.CUSTNO=WEEK1.CUSTNO AND CUST.STATE='CA'
            AND DATE BETWEEN '01/01/2000' And '01/07/2000'
            AND DATE IN ('01/07/2000','01/21/2000')
      GROUP BY WEEK1.CUSTNO
    UNION ALL
    SELECT WEEK3.CUSNTO, SUM(CHARGES), COUNT(CHARGES)
      FROM CUST,WEEK3
      WHERE CUST.CUSTNO=WEEK3 AND CUST.STATE='CA'
            AND DATE BETWEEN '01/15/2000' And '01/21/2000'
            AND DATE IN ('01/07/2000','01/21/2000')
      GROUP BY WEEK3.CUSTNO
  ) AS X(CUSTNO_U,SUM_U,CNT_U)
GROUP BY CUSNTO_U;
```

QBLOCKNO	PLANNO	TNAME	TABLE_TYPE	METHOD	QBLOCK TYPE	PARENT QBLOCKNO
1	1	DSNWFQB(02)	W	0		0
1	2			3		0
2	1			0	UNIONA	1
3	1	CUST	T	0		2
3	2	WEEK1	T	1		2
4	1	CUST	T	0		2
4	2	WEEK3	T	2		2

Figure 207. Plan table output for an example with a view with UNION ALLs

Performance of merge versus materialization

Merge performs better than materialization. For materialization, DB2 uses a table space scan to access the materialized temporary result. DB2 materializes a view or table expression only if it cannot merge.

As described above, materialization is a two-step process with the first step resulting in the formation of a temporary result. The smaller the temporary result, the more efficient is the second step. To reduce the size of the temporary result, DB2 attempts to evaluate certain predicates from the WHERE clause of the referencing statement at the first step of the process rather than at the second step. Only certain types of predicates qualify. First, the predicate must be a simple Boolean term predicate. Second, it must have one of the forms shown in Table 79.

Table 79. Predicate candidates for first-step evaluation

Predicate	Example
COL op literal	V1.C1 > hv1
COL IS (NOT) NULL	V1.C1 IS NOT NULL
COL (NOT) BETWEEN literal AND literal	V1.C1 BETWEEN 1 AND 10
COL (NOT) LIKE constant (ESCAPE constant)	V1.C2 LIKE 'p\%%' ESCAPE '\'

Note: Where ″op″ is =, <>, >, <, <=, or >=, and literal is either a host variable, constant, or special register. The literals in the BETWEEN predicate need not be identical.

Implied predicates generated through predicate transitive closure are also considered for first step evaluation.

Estimating a statement's cost

You can use EXPLAIN to populate a statement table, *owner*.DSN_STATEMNT_TABLE, at the same time as your PLAN_TABLE is being populated. DB2 provides cost estimates, in service units and in milliseconds, for SELECT, INSERT, UPDATE, and DELETE statements, both static and dynamic. The estimates do not take into account several factors, including cost adjustments that are caused by parallel processing, or the use of triggers or user-defined functions.

Use the information provided in the statement table to:
- Help you determine if a statement is not going to perform within range of your service-level agreements and to tune accordingly.

DB2 puts its cost estimate into one of two *cost categories*: category A or category B. Estimates that go into cost category A are the ones for which DB2 has adequate information to make an estimate. That estimate is not likely to be 100% accurate, but is likely to be more accurate than any estimate that is in cost category B.

DB2 puts estimates into cost category B when it is forced to use default values for its estimates, such as when no statistics are available, or because host variables are used in a query. See the description of the REASON column in Table 80 on page 713 for more information about how DB2 determines into which cost category an estimate goes.

- Give a system programmer a basis for entering service-unit values by which to govern dynamic statements.

  Information about using predictive governing is in Part 5 (Volume 2) of *DB2 Administration Guide*.

This section describes the following tasks to obtain and use cost estimate information from EXPLAIN:

1. "Creating a statement table"
2. "Populating and maintaining a statement table" on page 714
3. "Retrieving rows from a statement table" on page 714
4. "Understanding the implications of cost categories" on page 714

See Part 6 of *DB2 Application Programming and SQL Guide* for more information about how to change applications to handle the SQLCODES associated with predictive governing.

Creating a statement table

To collect information about a statement's estimated cost, create a table called DSN_STATEMNT_TABLE to hold the results of EXPLAIN. A copy of the statements that are needed to create the table are in the DB2 sample library, under the member name DSNTESC.

Figure 208 shows the format of a statement table.

```
CREATE TABLE DSN_STATEMNT_TABLE
  ( QUERYNO           INTEGER      NOT NULL WITH DEFAULT,
    APPLNAME          CHAR(8)      NOT NULL WITH DEFAULT,
    PROGNAME          CHAR(8)      NOT NULL WITH DEFAULT,
    COLLID            CHAR(18)     NOT NULL WITH DEFAULT,
    GROUP_MEMBER      CHAR(8)      NOT NULL WITH DEFAULT,
    EXPLAIN_TIME      TIMESTAMP    NOT NULL WITH DEFAULT,
    STMT_TYPE         CHAR(6)      NOT NULL WITH DEFAULT,
    COST_CATEGORY     CHAR(1)      NOT NULL WITH DEFAULT,
    PROCMS            INTEGER      NOT NULL WITH DEFAULT,
    PROCSU            INTEGER      NOT NULL WITH DEFAULT,
    REASON            VARCHAR(254) NOT NULL WITH DEFAULT);
```

Figure 208. Format of DSN_STATEMNT_TABLE

Table 80 on page 713 shows the content of each column. The first five columns of the DSN_STATEMNT_TABLE are the same as PLAN_TABLE.

Table 80. Descriptions of columns in DSN_STATEMNT_TABLE

Column Name	Description
QUERYNO	A number that identifies the statement being explained. See the description of the QUERYNO column in Table 74 on page 669 for more information. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.
APPLNAME	The name of the application plan for the row, or blank. See the description of the APPLNAME column in Table 74 on page 669 for more information.
PROGNAME	The name of the program or package containing the statement being explained, or blank. See the description of the PROGNAME column in Table 74 on page 669 for more information.
COLLID	The collection ID for the package, or blank. See the description of the COLLID column in Table 74 on page 669 for more information.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN, or blank. See the description of the GROUP_MEMBER column in Table 74 on page 669 for more information.
EXPLAIN_TIME	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.
STMT_TYPE	The type of statement being explained. Possible values are: **SELECT** SELECT **INSERT** INSERT **UPDATE** UPDATE **DELETE** DELETE **SELUPD** SELECT with FOR UPDATE OF **DELCUR** DELETE WHERE CURRENT OF CURSOR **UPDCUR** UPDATE WHERE CURRENT OF CURSOR
COST_CATEGORY	Indicates if DB2 was forced to use default values when making its estimates. Possible values: **A** Indicates that DB2 had enough information to make a cost estimate without using default values. **B** Indicates that some condition exists for which DB2 was forced to use default values. See the values in REASON to determine why DB2 was unable to put this estimate in cost category A.
PROCMS	The estimated processor cost, in milliseconds, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 milliseconds, which is equivalent to approximately 24.8 days. If the estimated value exceeds this maximum, the maximum value is reported.
PROCSU	The estimated processor cost, in service units, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 service units. If the estimated value exceeds this maximum, the maximum value is reported.

Table 80. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column Name	Description	
REASON	A string that indicates the reasons for putting an estimate into cost category B.	
	HAVING CLAUSE	A subselect in the SQL statement contains a HAVING clause.
	HOST VARIABLES	The statement uses host variables, parameter markers, or special registers.
	REFERENTIAL CONSTRAINTS	Referential constraints of the type CASCADE or SET NULL exist on the target table of a DELETE statement.
	TABLE CARDINALITY	The cardinality statistics are missing for one or more of the tables that are used in the statement.
	TRIGGERS	Triggers are defined on the target table of an INSERT, UPDATE, or DELETE statement.
	UDF	The statement uses user-defined functions.

Populating and maintaining a statement table

You populate a statement table at the same time as you populate the corresponding plan table. For more information, see "Populating and maintaining a plan table" on page 673.

Just as with the plan table, DB2 just adds rows to the statement table; it does not automatically delete rows. INSERT triggers are not activated unless you insert rows yourself using and SQL INSERT statement.

To clear the table of obsolete rows, use DELETE, just as you would for deleting rows from any table. You can also use DROP TABLE to drop a statement table completely.

Retrieving rows from a statement table

To retrieve all rows in a statement table, you can use a query like the following statement, which retrieves all rows about the statement that is represented by query number 13:

```
SELECT * FROM JOE.DSN_STATEMNT_TABLE
    WHERE QUERYNO = 13;
```

The QUERYNO, APPLNAME, PROGNAME, COLLID, and EXPLAIN_TIME columns contain the same values as corresponding columns of PLAN_TABLE for a given plan. You can use these columns to join the plan table and statement table:

```
SELECT A.*, PROCMS, COST_CATEGORY
 FROM JOE.PLAN_TABLE A, JOE.DSN_STATEMNT_TABLE B
    WHERE A.APPLNAME = 'APPL1' AND
    A.APPLNAME = B.APPLNAME AND
    A.PROGNAME = B.PROGNAME AND
    A.COLLID   = B.COLLID AND
    A.BIND_TIME = B.EXPLAIN_TIME
ORDER BY A.QUERYNO, A.QBLOCKNO, A.PLANNO, A.MIXOPSEQ;
```

Understanding the implications of cost categories

Cost categories are DB2's way of differentiating estimates for which adequate information is available from those for which it is not. You probably wouldn't want to

spend a lot of time tuning a query based on estimates that are returned in cost category B, because the actual cost could be radically different based on such things as what value is in a host variable, or how many levels of nested triggers and user-defined functions exist.

Similarly, if system administrators use these estimates as input into the resource limit specification table for governing (either predictive or reactive), they probably would want to give much greater latitude for statements in cost category B than for those in cost category A.

Because of the uncertainty involved, category B statements are also good candidates for reactive governing.

What goes into cost category B? DB2 puts a statement's estimate into cost category B when any of the following conditions exist:
- The statement has UDFs.
- Triggers are defined for the target table:
  - The statement is INSERT, and insert triggers are defined on the target table.
  - The statement is UPDATE, and update triggers are defined on the target table.
  - The statement is DELETE, and delete triggers are defined on the target table.
- The target table of a delete statement has referential constraints defined on it as the parent table, and the delete rules are either CASCADE or SET NULL.
- The WHERE clause predicate has one of the following forms:
  - COL op literal, and the literal is a host variable, parameter marker, or special register. The operator can be >, >=, <, <=, LIKE, or NOT LIKE.
  - COL BETWEEN literal AND literal where either literal is a host variable, parameter marker, or special register.
  - LIKE with an escape clause that contains a host variable.
- The cardinality statistics are missing for one or more tables that are used in the statement.
- A subselect in the SQL statement contains a HAVING clause.

What goes into cost category A? DB2 puts everything that doesn't fall into category B into category A.

Chapter 27. Parallel operations and query performance

When DB2 plans to access data from a table or index in a partitioned table space, it can initiate multiple parallel operations. The response time for data or processor-intensive queries can be significantly reduced.

Query I/O parallelism manages concurrent I/O requests for a single query, fetching pages into the buffer pool in parallel. This processing can significantly improve the performance of I/O-bound queries. I/O parallelism is used only when one of the other parallelism modes cannot be used.

Query CP parallelism enables true multi-tasking within a query. A large query can be broken into multiple smaller queries. These smaller queries run simultaneously on multiple processors accessing data in parallel. This reduces the elapsed time for a query.

To expand even farther the processing capacity available for processor-intensive queries, DB2 can split a large query across different DB2 members in a data sharing group. This is known as Sysplex query parallelism. For more information about Sysplex query parallelism, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

DB2 can use parallel operations for processing:
* Static and dynamic queries
* Local and remote data access
* Queries using single table scans and multi-table joins
* Access through an index, by table space scan or by list prefetch
* Sort operations

Parallel operations usually involve at least one table in a partitioned table space. Scans of large partitioned table spaces have the greatest performance improvements where both I/O and central processor (CP) operations can be carried out in parallel.

Parallelism for partitioned and nonpartitioned table spaces: Both partitioned and nonpartitioned table spaces can take advantage of query parallelism. Parallelism is now enabled to include non-clustering indexes. Thus, table access can be run in parallel when the application is bound with ANY and the table is accessed through a non-clustering index.

This chapter contains the following topics:

Comparing the methods of parallelism

The figures in this section show how the parallel methods compare with sequential prefetch and with each other. All three techniques assume access to a table space with three partitions, P1, P2, and P3. The notations P1, P2, and P3 are partitions of a table space. R1, R2, R3, and so on, are requests for sequential prefetch. The combination P2R1, for example, means the first request from partition 2.

Figure 209 shows **sequential processing**. With sequential processing, DB2 takes the 3 partitions in order, completing partition 1 before starting to process partition 2, and completing 2 before starting 3. Sequential prefetch allows overlap of CP processing with I/O operations, but I/O operations do not overlap with each other. In the example in Figure 209, a prefetch request takes longer than the time to process it. The processor is frequently waiting for I/O.
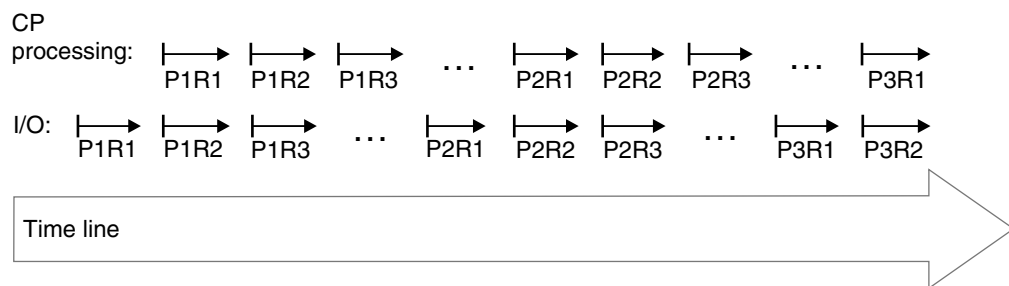
Figure 209. CP and I/O processing techniques. Sequential processing.

Figure 210 shows **parallel I/O operations**. With parallel I/O, DB2 prefetches data from the 3 partitions at one time. The processor processes the first request from each partition, then the second request from each partition, and so on. The processor is not waiting for I/O, but there is still only one processing task.
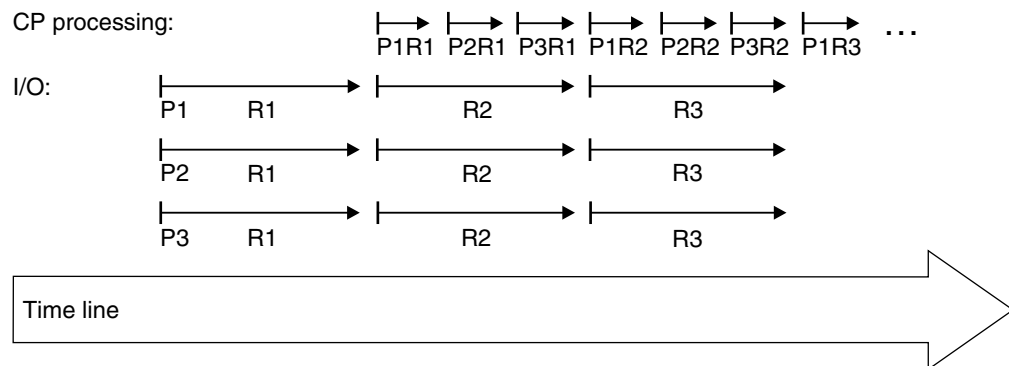
Figure 210. CP and I/O processing techniques. Parallel I/O processing.

Figure 211 on page 719 shows **parallel CP processing**. With CP parallelism, DB2 can use multiple parallel tasks to process the query. Three tasks working concurrently can greatly reduce the overall elapsed time for data-intensive and processor-intensive queries. The same principle applies for **Sysplex query parallelism**, except that the work can cross the boundaries of a single CPC.

CP task 1:
  P1R1   P1R2   P1R3   ...

I/O:
  P1R1   P1R2   P1R3   ...

CP task 2:
  P2R1   P2R2   P2R3   ...

I/O:
  P2R1   P2R2   P2R3   ...

CP task 3:
  P3R1   P3R2   P3R3   ...

I/O:
  P3R1   P3R2   P3R3   ...

Time line
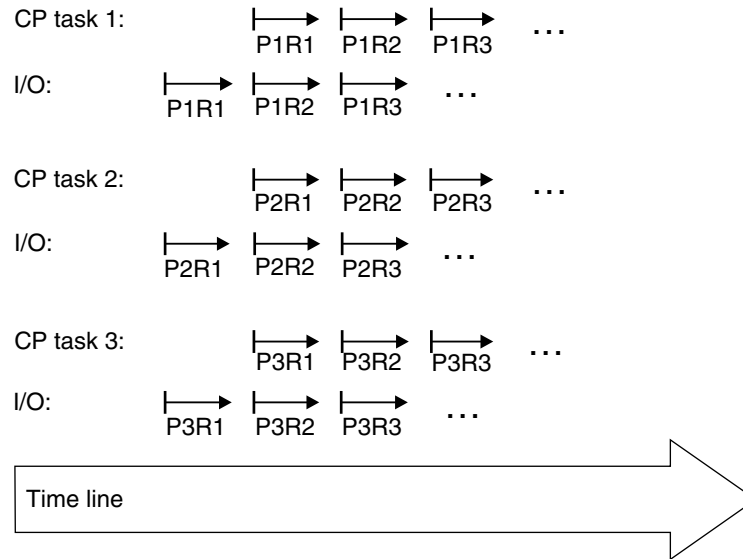
Figure 211. CP and I/O processing techniques. Query processing using CP parallelism. The tasks can be contained within a single CPC or can be spread out among the members of a data sharing group.

Queries that are most likely to take advantage of parallel operations: Queries that can take advantage of parallel processing are:

- Those in which DB2 spends most of the time fetching pages—an I/O-intensive query

  A typical I/O-intensive query is something like the following query, assuming that a table space scan is used on many pages:

  ```
  SELECT COUNT(*) FROM ACCOUNTS
   WHERE BALANCE > 0 AND
   DAYS_OVERDUE > 30;
  ```

- Those in which DB2 spends a lot of processor time and also, perhaps, I/O time, to process rows. Those include:

  - *Queries with intensive data scans and high selectivity.* Those queries involve large volumes of data to be scanned but relatively few rows that meet the search criteria.

  - *Queries containing aggregate functions.* Column functions (such as MIN, MAX, SUM, AVG, and COUNT) usually involve large amounts of data to be scanned but return only a single aggregate result.

  - *Queries accessing long data rows.* Those queries access tables with long data rows, and the ratio of rows per page is very low (one row per page, for example).

  - *Queries requiring large amounts of central processor time.* Those queries might be read-only queries that are complex, data-intensive, or that involve a sort.

    A typical processor-intensive query is something like:

    ```
    SELECT MAX(QTY_ON_HAND) AS MAX_ON_HAND,
      AVG(PRICE) AS AVG_PRICE,
      AVG(DISCOUNTED_PRICE) AS DISC_PRICE,
      SUM(TAX) AS SUM_TAX,
      SUM(QTY_SOLD) AS SUM_QTY_SOLD,
      SUM(QTY_ON_HAND - QTY_BROKEN) AS QTY_GOOD,
      AVG(DISCOUNT) AS AVG_DISCOUNT,
      ORDERSTATUS,
      COUNT(*) AS COUNT_ORDERS
    ```

```
        FROM  ORDER_TABLE
  WHERE SHIPPER = 'OVERNIGHT' AND
        SHIP_DATE < DATE('1996-01-01')
  GROUP BY ORDERSTATUS
  ORDER BY ORDERSTATUS;
```

Terminology: When the term *task* is used with information on parallel processing, the context should be considered. For parallel query CP processing or Sysplex query parallelism, task is an actual MVS execution unit used to process a query. For parallel I/O processing, a task simply refers to the processing of one of the concurrent I/O streams.

A **parallel group** is the term used to name a particular set of parallel operations (parallel tasks or parallel I/O operations). A query can have more than one parallel group, but each parallel group within the query is identified by its own unique ID number.

The **degree of parallelism** is the number of parallel tasks or I/O operations that DB2 determines can be used for the operations on the parallel group.

Enabling parallel processing

Queries can only take advantage of parallelism if you enable parallel processing. Use the following actions to enable parallel processing:

- For **static SQL**, specify DEGREE(ANY) on BIND or REBIND. This bind option affects static SQL only and does not enable parallelism for dynamic statements.
- For **dynamic SQL**, set the CURRENT DEGREE special register to 'ANY'. Setting the special register affects dynamic statements only. It will have no effect on your static SQL statements. You should also make sure that parallelism is not disabled for your plan, package, or authorization ID in the RLST. You can set the special register with the following SQL statement:

  ```
  SET CURRENT DEGREE='ANY';
  ```

  It is also possible to change the special register default from 1 to ANY for the entire DB2 subsystem by modifying the CURRENT DEGREE field on installation panel DSNTIP4.

- If you bind with isolation CS, choose also the option CURRENTDATA(NO), if possible. This option can improve performance in general, but it also ensures that DB2 will consider parallelism for ambiguous cursors. If you bind with CURRENDATA(YES) and DB2 cannot tell if the cursor is read-only, DB2 does not consider parallelism. It is best to always indicate when a cursor is read-only by indicating FOR FETCH ONLY or FOR READ ONLY on the DECLARE CURSOR statement.

- The virtual buffer pool parallel sequential threshold (VPPSEQT) value must be large enough to provide adequate buffer pool space for parallel processing. For a description of buffer pools and thresholds, see Part 5 (Volume 2) of *DB2 Administration Guide*.

If you enable parallel processing when DB2 estimates a given query's I/O and central processor cost is high, multiple parallel tasks can be activated if DB2 estimates that elapsed time can be reduced by doing so.

Special requirements for CP parallelism: DB2 must be running on a central processor complex that contains two or more tightly-coupled processors (sometimes called central processors, or CPs). If only one CP is online when the query is bound, DB2 considers only parallel I/O operations.

DB2 also considers only parallel I/O operations if you declare a cursor WITH HOLD and bind with isolation RR or RS. For further restrictions on parallelism, see Table 81.

For complex queries, run the query in parallel within a member of a data sharing group. With Sysplex query parallelism, use the power of the data sharing group to process individual complex queries on many members of the data sharing group. For more information on how you can use the power of the data sharing group to run complex queries, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

Limiting the degree of parallelism: If you want to limit the maximum number of parallel tasks that DB2 generates, you can use the installation parameter MAX DEGREE in the DSNTIP4 panel. Changing MAX DEGREE, however, is not the way to turn parallelism off. You use the DEGREE bind parameter or CURRENT DEGREE special register to turn parallellism off.

When parallelism is not used

Parallelism is not used for all queries; for some access paths, it doesn't make sense to incur parallelism overhead. If you are selecting from a temporary table, you won't get parallelism for that, either. If you are not getting parallelism, check Table 81 to see if your query uses any of the access paths that do not allow parallelism.

Table 81. Checklist of parallel modes and query restrictions

If query uses this...	Is parallelism allowed?			Comments
	I/O	**CP**	**Sysplex**	
Access via RID list (list prefetch and multiple index access)	Yes	Yes	No	Indicated by an "L" in the PREFETCH column of PLAN_TABLE, or an M, MX, MI, or MQ in the ACCESSTYPE column of PLAN_TABLE.
Queries that return LOB values	Yes	Yes	No	
Merge scan join on more than one column	No	No	No	
Queries that qualify for direct row access	No	No	No	Indicated by D in the PRIMARY_ACCESS_TYPE column of PLAN_TABLE
Materialized views or materialized nested table expressions at reference time.	No	No	No	
EXISTS within WHERE predicate	No	No	No	

DB2 avoids certain hybrid joins when parallelism is enabled: To ensure that you can take advantage of parallelism, DB2 does not pick one type of hybrid join (SORTN_JOIN=Y) when the plan or package is bound with CURRENT DEGREE=ANY or if the CURRENT DEGREE special register is set to 'ANY'.

Interpreting EXPLAIN output

To understand how DB2 plans to use parallelism, examine your PLAN_TABLE output. (Details on all columns in PLAN_TABLE are described in Table 74 on page 669. This section describes a method for examining PLAN_TABLE columns for parallelism and gives several examples.

A method for examining PLAN_TABLE columns for parallelism

The steps for interpreting the output for parallelism are as follows:

1. **Determine if DB2 plans to use parallelism:**

   For each query block (QBLOCKNO) in a query (QUERYNO), a non-null value in ACCESS_DEGREE or JOIN_DEGREE indicates that some degree of parallelism is planned.

2. **Identify the parallel groups in the query:**

   All steps (PLANNO) with the same value for ACCESS_PGROUP_ID, JOIN_PGROUP_ID, SORTN_PGROUP_ID, or SORTC_PGROUP_ID indicate that a set of operations are in the same parallel group. Usually, the set of operations involves various types of join methods and sort operations. Parallel group IDs can appear in the same row of PLAN_TABLE output, or in different rows, depending on the operation being performed. The examples in "PLAN_TABLE examples showing parallelism" help clarify this concept.

3. **Identify the parallelism mode:**

   The column PARALLELISM_MODE tells you the kind of parallelism that is planned (I, C, or X). Within a query block, you cannot have a mixture of "I" and "C" parallel modes. However, a statement that uses more than one query block, such as a UNION, can have "I" for one query block and "C" for another. It is possible to have a mixture of "C" and "X" modes in a query block but not in the same parallel group.

   If the statement was bound while this DB2 is a member of a data sharing group, the PARALLELISM_MODE column can contain "X" even if only this one DB2 member is active. This lets DB2 take advantage of extra processing power that might be available at execution time. If other members are not available at execution time, then DB2 runs the query within the single DB2 member.

PLAN_TABLE examples showing parallelism

For these examples, the other values would not change whether the PARALLELISM_MODE is I, C, or X.

- **Example 1: single table access**

  Assume that DB2 decides at bind time to initiate three concurrent requests to retrieve data from table T1. Part of PLAN_TABLE appears as follows. If DB2 decides not to use parallel operations for a step, ACCESS_DEGREE and ACCESS_PGROUP_ID contain null values.

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)

- **Example 2: nested loop join**

  Consider a query that results in a series of nested loop joins for three tables, T1, T2 and T3. T1 is the outermost table, and T3 is the innermost table. DB2 decides at bind time to initiate three concurrent requests to retrieve data from each of the

three tables. For the nested loop join method, all the retrievals are in the same parallel group. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	1	3	1	3	1	(null)	(null)
T3	1	3	1	3	1	(null)	(null)

- **Example 3: merge scan join**

  Consider a query that causes a merge scan join between two tables, T1 and T2. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. The scan and sort of T1 occurs in one parallel group. The scan and sort of T2 occurs in another parallel group. Furthermore, the merging phase can potentially be done in parallel. Here, a third parallel group is used to initiate three concurrent requests on each intermediate sorted table. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	2	6	2	3	3	1	2

- **Example 4: hybrid join**

  Consider a query that results in a hybrid join between two tables, T1 and T2. Furthermore, T1 needs to be sorted; as a result, in PLAN_TABLE the T2 row has SORTC_JOIN=Y. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. Parallel operations are used for a join through a clustered index of T2.

  Because T2's RIDs can be retrieved by initiating concurrent requests on the partitioned index, the joining phase is a parallel step. The retrieval of T2's RIDs and T2's rows are in the same parallel group. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	4	6	2	6	2	1	(null)

Tuning parallel processing

Much of the information in this section applies also to Sysplex query parallelism. See Chapter 6 of *DB2 Data Sharing: Planning and Administration* for more information.

It is possible for a parallel group run at a parallel degree less than that shown in the PLAN_TABLE output. The following can cause a reduced degree of parallelism:

- Buffer pool availability
- Logical contention.

Consider a nested loop join. The inner table could be in a partitioned or nonpartitioned table space, but DB2 is more likely to use a parallel join operation when the outer table is partitioned.

- Physical contention
- Run time host variables

A host variable can determine the qualifying partitions of a table for a given query. In such cases, DB2 defers the determination of the planned degree of parallelism until run time, when the host variable value is known.

- Updatable cursor

At run time, DB2 might determine that an ambiguous cursor is updatable.

- A change in the configuration of online processors

If fewer processors are online at run time, DB2 might need to reformulate the parallel degree.

Locking considerations for repeatable read applications: For CP parallelism, locks are obtained independently by each task. Be aware that this can possibly increase the total number of locks taken for applications that:

- Use an isolation level of repeatable read
- Use CP parallelism
- Repeatedly access the table space using a lock mode of IS without issuing COMMITs

As is recommended for all repeatable-read applications, be sure to issue frequent COMMITs to release the lock resources that are held. Repeatable read or read stability isolation cannot be used with Sysplex query parallelism.

Disabling query parallelism

To disable parallel operations, do any of the following actions:

- For static SQL, rebind to change the option DEGREE(ANY) to DEGREE(1). You can do this by using the DB2I panels, the DSN subcommands, or the DSNH CLIST. The default is DEGREE(1).
- For dynamic SQL, execute the following SQL statement:

```
SET CURRENT DEGREE = '1';
```

The default value for CURRENT DEGREE is 1 unless your installation has changed the default for the CURRENT DEGREE special register.

System controls can be used to disable parallelism, as well. These are described in Part 5 (Volume 2) of *DB2 Administration Guide*.

Chapter 28. Programming for the Interactive System Productivity Facility (ISPF)

The Interactive System Productivity Facility (ISPF) helps you to construct and execute dialogs. DB2 includes a sample application that illustrates how to use ISPF through the call attachment facility (CAF). Instructions for compiling, printing, and using the application are in Part 2 of *DB2 Installation Guide*. This chapter describes how to structure applications for use with ISPF.

The following sections discuss scenarios for interaction among your program, DB2, and ISPF. Each has advantages and disadvantages in terms of efficiency, ease of coding, ease of maintenance, and overall flexibility.

Using ISPF and the DSN command processor

There are some restrictions on how you make and break connections to DB2 in any structure. If you use the PGM option of ISPF SELECT, ISPF passes control to your load module by the LINK macro; if you use CMD, ISPF passes control by the ATTACH macro.

The DSN command processor (see "DSN command processor" on page 421) permits only single task control block (TCB) connections. Take care not to change the TCB after the first SQL statement. ISPF SELECT services change the TCB if you started DSN under ISPF, so you cannot use these to pass control from load module to load module. Instead, use LINK, XCTL, or LOAD.

Figure 212 on page 726 shows the task control blocks that result from attaching the DSN command processor below TSO or ISPF.

If you are in ISPF and running under DSN, you can perform an ISPLINK to another program, which calls a CLIST. In turn, the CLIST uses DSN and another application. Each such use of DSN creates a separate unit of recovery (process or transaction) in DB2.

All such initiated DSN work units are unrelated, with regard to isolation (locking) and recovery (commit). It is possible to deadlock with yourself; that is, one unit (DSN) can request a serialized resource (a data page, for example) that another unit (DSN) holds incompatibly.

A COMMIT in one program applies only to that process. There is no facility for coordinating the processes.
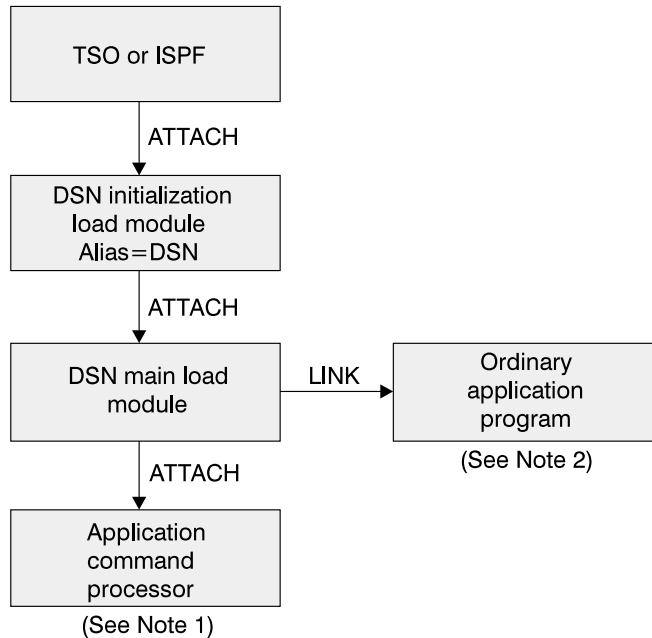
Figure 212. DSN task structure. Each block represents a task control block (TCB).

Notes to Figure 212:
1. The RUN command with the CP option causes DSN to attach your program and create a new TCB.
2. The RUN command without the CP option causes DSN to link to your program.

Invoking a single SQL program through ISPF and DSN

With this structure, the user of your application first invokes ISPF, which displays the data and selection panels. When the user selects the program on the selection panel, ISPF calls a CLIST that runs the program. A corresponding CLIST might contain:

```
DSN
    RUN PROGRAM(MYPROG) PLAN(MYPLAN)
END
```

The application has one large load module and one plan.

Disadvantages: For large programs of this type, you want a more modular design, making the plan more flexible and easier to maintain. If you have one large plan, you must rebind the entire plan whenever you change a module that includes SQL statements. [2] You cannot pass control to another load module that makes SQL calls by using ISPLINK; rather, you must use LINK, XCTL, or LOAD and BALR.

If you want to use ISPLINK, then call ISPF to run under DSN:

```
DSN
    RUN PROGRAM(ISPF) PLAN(MYPLAN)
END
```

You then have to leave ISPF before you can start your application.

---

2. To achieve a more modular construction when all parts of the program use SQL, consider using packages. See "Chapter 16. Planning for DB2 program preparation" on page 313.

Furthermore, the entire program is dependent on DB2; if DB2 is not running, no part of the program can begin or continue to run.

Invoking multiple SQL programs through ISPF and DSN

You can break a large application into several different functions, each communicating through a common pool of shared variables controlled by ISPF. You might write some functions as separately compiled and loaded programs, others as EXECs or CLISTs. You can start any of those programs or functions through the ISPF SELECT service, and you can start that from a program, a CLIST, or an ISPF selection panel.

When you use the ISPF SELECT service, you can specify whether ISPF should create a new ISPF variable pool before calling the function. You can also break a large application into several independent parts, each with its own ISPF variable pool.

You can call different parts of the program in different ways. For example, you can use the PGM option of ISPF SELECT:

```
PGM(program-name) PARM(parameters)
```

Or, you can use the CMD option:

```
CMD(command)
```

For a part that accesses DB2, the command can name a CLIST that starts DSN:

```
DSN
    RUN PROGRAM(PART1) PLAN(PLAN1) PARM(input from panel)
END
```

Breaking the application into separate modules makes it more flexible and easier to maintain. Furthermore, some of the application might be independent of DB2; portions of the application that do not call DB2 can run, even if DB2 is not running. A stopped DB2 database does not interfere with parts of the program that refer only to other databases.

Disadvantages: The modular application, on the whole, has to do more work. It calls several CLISTs, and each one must be located, loaded, parsed, interpreted, and executed. It also makes and breaks connections to DB2 more often than the single load module. As a result, you might lose some efficiency.

Invoking multiple SQL programs through ISPF and CAF

You can use the call attachment facility (CAF) to call DB2; for details, see "Chapter 29. Programming for the call attachment facility (CAF)" on page 729. The ISPF/CAF sample connection manager programs (DSN8SPM and DSN8SCM) take advantage of the ISPLINK SELECT services, letting each routine make its own connection to DB2 and establish its own thread and plan.

With the same modular structure as in the previous example, using CAF is likely to provide greater efficiency by reducing the number of CLISTs. This does not mean, however, that any DB2 function executes more quickly.

Disadvantages: Compared to the modular structure using DSN, the structure using CAF is likely to require a more complex program, which in turn might require assembler language subroutines. For more information, see "Chapter 29. Programming for the call attachment facility (CAF)" on page 729.

Chapter 29. Programming for the call attachment facility (CAF)

An attachment facility is a part of the DB2 code that allows other programs to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. With the call attachment facility (CAF), your application program can establish and control its own connection to DB2. Programs that run in MVS batch, TSO foreground, and TSO background can use CAF.

It is also possible for IMS batch applications to access DB2 databases through CAF, though that method does not coordinate the commitment of work between the IMS and DB2 systems. We highly recommend that you use the DB2 DL/I batch support for IMS batch applications.

CICS application programs must use the CICS attachment facility; IMS application programs, the IMS attachment facility. Programs running in TSO foreground or TSO background can use either the DSN command processor or CAF; each has advantages and disadvantages.

Prerequisite knowledge: Analysts and programmers who consider using CAF must be familiar with MVS concepts and facilities in the following areas:
- The CALL macro and standard module linkage conventions
- Program addressing and residency options (AMODE and RMODE)
- Creating and controlling tasks; multitasking
- Functional recovery facilities such as ESTAE, ESTAI, and FRRs
- Asynchronous events and TSO attention exits (STAX)
- Synchronization techniques such as WAIT/POST.

Call attachment facility capabilities and restrictions

To decide whether to use the call attachment facility, consider the capabilities and restrictions described on the pages following.

Capabilities when using CAF

A program using CAF can:
- Access DB2 from MVS address spaces where TSO, IMS, or CICS do not exist.
- Access DB2 from multiple MVS tasks in an address space.
- Access the DB2 IFI.
- Run when DB2 is down (though it cannot run SQL when DB2 is down).
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor (or of any DB2 code).
- Run above or below the 16-megabyte line. (The CAF code resides below the line.)
- Establish an *explicit* connection to DB2, through a *CALL* interface, with control over the exact state of the connection.
- Establish an *implicit* connection to DB2, by using SQL statements or IFI calls without first calling CAF, with a default plan name and subsystem identifier.
- Verify that your application is using the correct release of DB2.
- Supply event control blocks (ECBs), for DB2 to post, that signal start-up or termination.

- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages as desired.

Task capabilities

Any task in an address space can establish a connection to DB2 through CAF. There can be only one connection for each task control block (TCB). A DB2 service request issued by a programs running under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Each connected task can run a plan. Multiple tasks in a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without fully breaking its connection to DB2.

CAF does not generate task structures, nor does it provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines and not Enabled Unlocked Task (EUT) FRR routines.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Your application design must consider that possibility.

Programming language

You can write CAF applications in assembler language, C, COBOL, FORTRAN, and PL/I. When choosing a language to code your application in, consider these restrictions:

- If you need to use MVS macros (ATTACH, WAIT, POST, and so on), you must choose a programming language that supports them or else embed them in modules written in assembler language.
- The CAF TRANSLATE function is not available from FORTRAN. To use the function, code it in a routine written in another language, and then call that routine from FORTRAN.

You can find a sample assembler program (DSN8CA) and a sample COBOL program (DSN8CC) that use the call attachment facility in library *prefix*.SDSNSAMP. A PL/I application (DSN8SPM) calls DSN8CA, and a COBOL application (DSN8SCM) calls DSN8CC. For more information on the sample applications and on accessing the source code, see "Appendix B. Sample applications" on page 829.

Tracing facility

A tracing facility provides diagnostic messages that aid in debugging programs and diagnosing errors in the CAF code. In particular, attempts to use CAF incorrectly cause error messages in the trace stream.

Program preparation

Preparing your application program to run in CAF is similar to preparing it to run in other environments, such as CICS, IMS, and TSO. You can prepare a CAF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST. For examples and guidance in program preparation, see "Chapter 20. Preparing an application program to run" on page 395.

CAF requirements

When you write programs that use CAF, be aware of the following characteristics.

Program size
The CAF code requires about 16K of virtual storage per address space and an additional 10K for each TCB using CAF.

Use of LOAD
CAF uses MVS SVC LOAD to load two modules as part of the initialization following your first service request. Both modules are loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces the LOAD SVC, then you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard MVS LOAD macro.

Using CAF in IMS batch
If you use CAF from IMS batch, you must write data to only one system in any one unit of work. If you write to both systems within the same unit, a system failure can leave the two databases inconsistent with no possibility of automatic recovery. To end a unit of work in DB2, execute the SQL COMMIT statement; to end one in IMS, issue the SYNCPOINT command.

Run environment
Applications requesting DB2 services must adhere to several run environment characteristics. Those characteristics must be in effect regardless of the attachment facility you use. They are not unique to CAF.

- The application must be running in TCB mode. SRB mode is not supported.
- An application task cannot have any EUT FRRs active when requesting DB2 services. If an EUT FRR is active, DB2's functional recovery can fail, and your application can receive some unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. Therefore:
  - An application must not use CAF in an CICS or IMS address space.
  - An application that runs in an address space that has a CAF connection to DB2 cannot connect to DB2 using RRSAF.
  - An application that runs in an address space that has an RRSAF connection to DB2 cannot connect to DB2 using CAF.
  - An application cannot invoke the MVS AXSET macro after executing the CAF CONNECT call and before executing the CAF DISCONNECT call.
- One attachment facility cannot start another. This means that your CAF application cannot use DSN, and a DSN RUN subcommand cannot call your CAF application.
- The language interface module for CAF, DSNALI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load CAF below the 16MB line, you must link-edit DSNALI again.

Running DSN applications under CAF
It is possible, though not recommended, to run existing DSN applications with CAF merely by allowing them to make implicit connections to DB2. For DB2 to make an implicit connection successfully, the plan name for the application must be the same as the member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call. You must also substitute the DSNALI language interface module for the TSO language interface module, DSNELI.

There is no significant advantage to running DSN applications with CAF, and the loss of DSN services can affect how well your program runs. We do not recommend that you run DSN applications with CAF unless you provide an application controller

to manage the DSN application and replace any needed DSN functions. Even then, you could have to change the application to communicate connection failures to the controller correctly.

How to use CAF

To use CAF, you must first make available a load module known as the *call attachment language interface*, or DSNALI. For considerations for loading or link-editing this module, see "Accessing the CAF language interface" on page 735.

When the language interface is available, your program can make use of the CAF in two ways:

- Implicitly, by including SQL statements or IFI calls in your program just as you would in any program. The CAF facility establishes the connections to DB2 using default values for the pertinent parameters described under "Implicit connections" on page 734.
- Explicitly, by writing CALL DSNALI statements, providing the appropriate options. For the general form of the statements, see "CAF function descriptions" on page 737.

The first element of each option list is a *function*, which describes the action you want CAF to take. The available values of *function* and an approximation of their effects, see "Summary of connection functions" on page 733. The effect of any function depends in part on what functions the program has already run. Before using any function, be sure to read the description of its usage. Also read "Summary of CAF behavior" on page 749, which describes the influence of previous functions.

You might possibly structure a CAF configuration like this one:
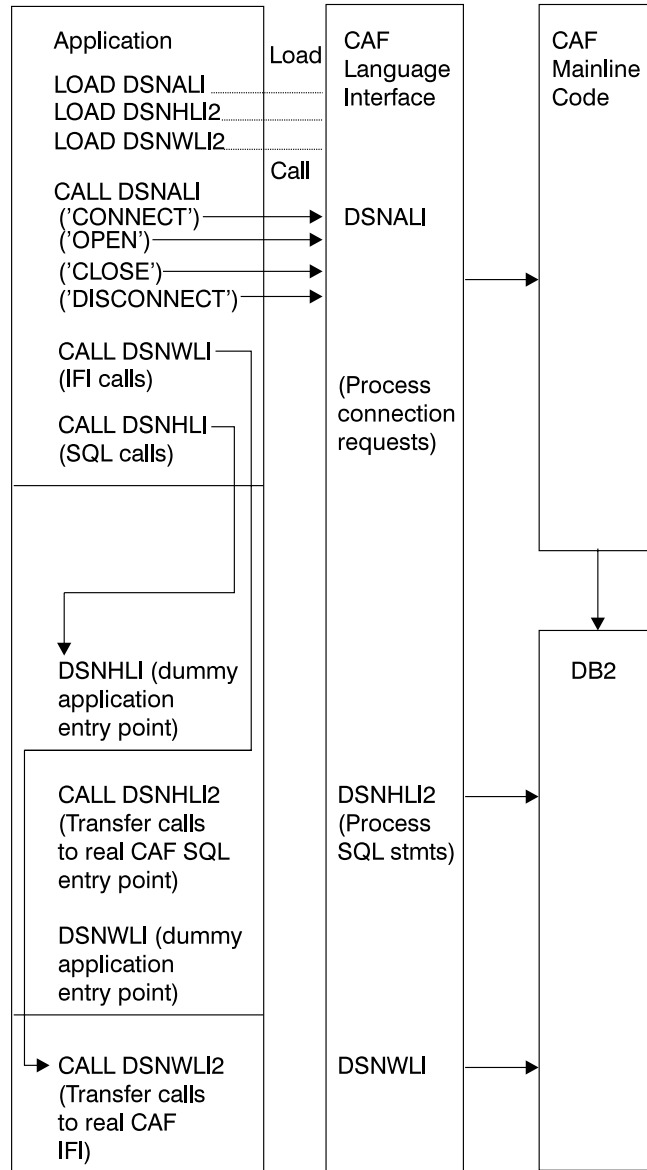
Figure 213. Sample call attachment facility configuration

The remainder of this chapter discusses:
- Summary of connection functions
- "Sample scenarios" on page 750
- "Exits from your application" on page 751
- "Error messages and dsntrace" on page 752
- "Program examples" on page 753.

Summary of connection functions

You can use the following functions with CALL DSNALI:

CONNECT
Establishes the task (TCB) as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is also initialized as a user of DB2. See "CONNECT: Syntax and usage" on page 739.

OPEN

Allocates a DB2 plan. You must allocate a plan before DB2 can process SQL statements. If you did not request the CONNECT function, OPEN implicitly establishes the task, and optionally the address space, as a user of DB2. See "OPEN: Syntax and usage" on page 743.

CLOSE

Optionally commits or abends any database changes and deallocates the plan. If OPEN implicitly requests the CONNECT function, CLOSE removes the task, and possibly the address space, as a user of DB2. See "CLOSE: Syntax and usage" on page 745.

DISCONNECT

Removes the task as a user of DB2 and, if this is the last or only task in the address space with a DB2 connection, terminates the address space connection to DB2. See "DISCONNECT: Syntax and usage" on page 746.

TRANSLATE

Returns an SQLCODE and printable text in the SQLCA that describes a DB2 hexadecimal error reason code. See "TRANSLATE: Syntax and usage" on page 747. You cannot call the TRANSLATE function from the FORTRAN language.

Implicit connections

If you do not explicitly specify executable SQL statements in a CALL DSNALI statement of your CAF application, CAF initiates implicit CONNECT and OPEN requests to DB2. Although CAF performs these connection requests using the default values defined below, the requests are subject to the same DB2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

Subsystem name

The default name specified in the module DSNHDECP. CAF uses the installation default DSNHDECP, unless your own DSNHDECP is in a library in a STEPLIB of JOBLIB concatenation, or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

Plan name

The member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call. If your program can make its first SQL call from different modules with different DBRMs, then you cannot use a default plan name; you must use an explicit call using the OPEN function.

If your application includes *both* SQL and IFI calls, you must issue *at least one* SQL call before you issue any IFI calls. This ensures that your application uses the correct plan.

There are different types of implicit connections. The simplest is for application to run neither CONNECT nor OPEN. You can also use CONNECT only or OPEN only. Each of these implicitly connects your application to DB2. To terminate an implicit connection, you must use the proper calls. See Table 87 on page 749 for details.

Your application program must successfully connect, either implicitly or explicitly, to DB2 before it can execute any SQL calls to the CAF DSNHLI entry point. Therefore, the application program must first determine the success or failure of all implicit connection requests.

For implicit connection requests, register 15 contains the return code and register 0 contains the reason code. The return code and reason code are also in the message text for SQLCODE -991. The application program should examine the return and reason codes immediately after the first executable SQL statement within the application program. There are two ways to do this:

- Examine registers 0 and 15 directly.
- Examine the SQLCA, and if the SQLCODE is -991, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection was successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

Accessing the CAF language interface

Part of the call attachment facility is a DB2 load module, DSNALI, known as the *call attachment facility language interface*. DSNALI has the alias names DSNHLI2 and DSNWLI2. The module has five entry points: DSNALI, DSNHLI, DSNHLI2, DSNWLI, and DSNWLI2:

- Entry point DSNALI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLI2 handle SQL calls (use DSNHLI if your application program link-edits CAF; use DSNHLI2 if your application program loads CAF).
- DSNWLI and DSNWLI2 handle IFI calls (use DSNWLI if your application program link-edits CAF; use DSNWLI2 if your application program loads CAF).

You can access the DSNALI module by either explicitly issuing LOAD requests when your program runs, or by including the module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Explicit load of DSNALI

To load DSNALI, issue MVS LOAD service requests for entry points DSNALI and DSNHLI2. If you use IFI services, you must also load DSNWLI2. The entry point addresses that LOAD returns are saved for later use with the CALL macro.

By explicitly loading the DSNALI module, you beneficially isolate the maintenance of your application from future IBM service to the language interface. If the language interface changes, the change will probably not affect your load module.

You must indicate to DB2 which entry point to use. You can do this in one of two ways:

- Specify the precompiler option ATTACH(CAF).

  This causes DB2 to generate calls that specify entry point DSNHLI2. You cannot use this option if your application is written in FORTRAN.
- Code a dummy entry point named DSNHLI within your load module.

  If you do not specify the precompiler option ATTACH, the DB2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code corresponding to the dummy entry point must preserve the option list passed in R1 and call DSNHLI2 specifying the same option list. For a coding example of a dummy DSNHLI entry point, see "Using dummy entry point DSNHLI" on page 759.

Link-editing DSNALI

You can include the CAF language interface module DSNALI in your load module during a link-edit step. The module must be in a load module library, which is included either in the SYSLIB concatenation or another INCLUDE library defined in the linkage editor JCL. Because all language interface modules contain an entry point declaration for DSNHLI, the linkage editor JCL must contain an INCLUDE linkage editor control statement for DSNALI; for example, INCLUDE DB2LIB(DSNALI). By coding these options, you avoid inadvertently picking up the wrong language interface module.

If you do not need explicit calls to DSNALI for CAF functions, including DSNALI in your load module has some advantages. When you include DSNALI during the link-edit, you need not code the previously described dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNALI contains an entry point for DSNHLI, which is identical to DSNHLI2, and an entry point DSNWLI, which is identical to DSNWLI2.

A disadvantage to link-editing DSNALI into your load module is that any IBM service to DSNALI requires a new link-edit of your load module.

General properties of CAF connections

Some of the basic properties of the connection the call attachment facility makes with DB2 are:

- **Connection name**: DB2CALL. You can use the DISPLAY THREAD command to list CAF applications having the connection name DB2CALL.
- **Connection type**: BATCH. BATCH connections use a single phase commit process coordinated by DB2. Application programs can also use the SQL COMMIT and ROLLBACK statements.
- **Authorization IDs**: DB2 establishes authorization identifiers for each task's connection when it processes the connection for each task. For the BATCH connection type, DB2 creates a list of authorization IDs based upon the authorization ID associated with the address space and the list is the *same* for every task. A location can provide a DB2 connection authorization exit routine to change the list of IDs. For information about authorization IDs and the connection authorization exit routine, see Appendix B (Volume 2) of *DB2 Administration Guide*.
- **Scope**: The CAF processes connections as if each task is entirely isolated. When a task requests a function, the CAF passes the functions to DB2, unaware of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem are aware of the connection status of multiple tasks in an address space.

Task termination

If a connected task terminates normally before the CLOSE function deallocates the plan, then DB2 commits any database changes that the thread made since the last commit point. If a connected task abends before the CLOSE function deallocates the plan, then DB2 rolls back any database changes since the last commit point.

In either case, DB2 deallocates the plan, if necessary, and terminates the task's connection before it allows the task to terminate.

DB2 abend

If DB2 abends while an application is running, the application is rolled back to the last commit point. If DB2 terminates while processing a commit request, DB2 either

commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

CAF function descriptions

To code CAF functions in C, COBOL, FORTRAN, or PL/I, follow the individual language's rules for making calls to assembler routines. Specify the return code and reason code parameters in the parameter list for each CAF call.

A description of the call attach register and parameter list conventions for assembler language follow. Following it, the syntax description of specific functions describe the parameters for those particular functions.

Register conventions

If you do not specify the return code and reason code parameters in your CAF calls, CAF puts a return code in register 15 and a reason code in register 0. CAF also supports high-level languages that cannot interrogate individual registers. See Figure 214 on page 738 and the discussion following it for more information. The contents of registers 2 through 14 are preserved across calls. You must conform to the following standard calling conventions:

Register	Usage
R1	Parameter list pointer (for details, see "Call DSNALI parameter list")
R13	Address of caller's save area
R14	Caller's return address
R15	CAF entry point address

Call DSNALI parameter list

Use a standard MVS CALL parameter list. Register 1 points to a list of fullword addresses that point to the actual parameters. The last address *must* contain a 1 in the high-order bit. Figure 214 on page 738 shows a sample parameter list structure for the CONNECT function.

When you code CALL DSNALI statements, you must specify all parameters that come before *Return Code*. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters for explicit connection service requests. Defaults are provided only for implicit connections.

All parameters starting with *Return Code* are optional.

For all languages except assembler language, code zero for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are coding a CONNECT call in a COBOL program. You want to specify all parameters except *Return Code*. Write the call in this way:

```
CALL 'DSNALI' USING FUNCTN SSID TECB SECB RIBPTR
   BY CONTENT ZERO BY REFERENCE REASCODE SRDURA EIBPTR.
```

For an assembler language call, code a comma for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, code a CONNECT call like this to specify all optional parameters except *Return Code*:

```
CALL  DSNALI,(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,,REASCODE,SRDURA,EIBPTR,GROUPOVERRIDE)
```
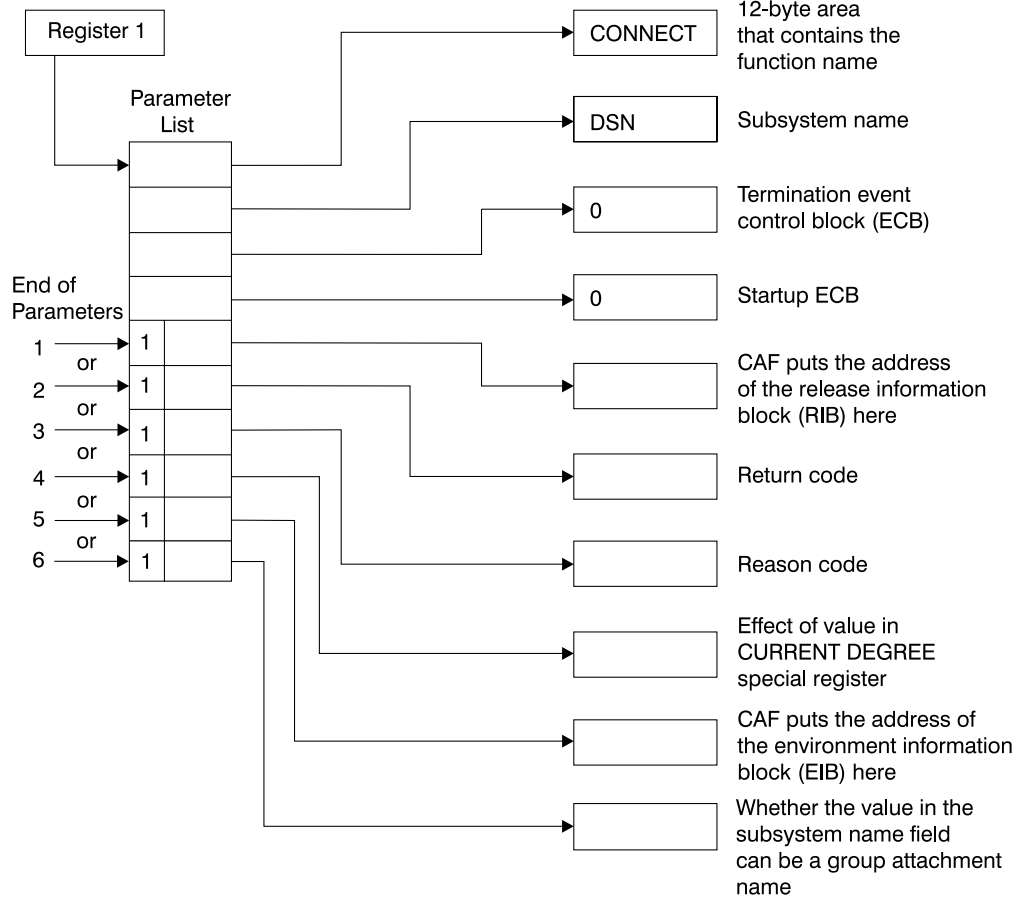
Figure 214. The parameter list for a CONNECT call

Figure 214 illustrates how you can use the indicator '*end of parameter list*' to control the return codes and reason code fields following a CAF CONNECT call. Each of the three illustrated termination points apply to all CAF parameter lists:

1. Terminates the parameter list without specifying the parameters *retcode, reascode*, and *srdura*, and places the return code in register 15 and the reason code in register 0.

   Terminating at this point ensures compatibility with CAF programs that require a return code in register 15 and a reason code in register 0.

2. Terminates the parameter list after the return code field, and places the return code in the parameter list and the reason code in register 0.

   Terminating at this point permits the application program to take action, based on the return code, without further examination of the associated reason code.

3. Terminates the parameter list after the reason code field and places the return code and the reason code in the parameter list.

   Terminating at this point provides support to high-level languages that are unable to examine the contents of individual registers.

   If you code your CAF application in assembler language, you can specify this parameter and omit the return code parameter. To do this, specify a comma as a place-holder for the omitted return code parameter.

4. Terminates the parameter list after the parameter *srdura*.

If you code your CAF application in assembler language, you can specify this parameter and omit the return code and reason code parameters. To do this, specify commas as place-holders for the omitted parameters.

5. Terminates the parameter list after the parameter *eibptr*.

   If you code your CAF application in assembler language, you can specify this parameter and omit the return code, reason code, or srdura parameters. To do this, specify commas as place-holders for the omitted parameters.

6. Terminates the parameter list after the parameter *groupoverride*.

   If you code your CAF application in assembler language, you can specify this parameter and omit the return code, reason code, srdura, or eibptr parameters. To do this, specify commas as place-holders for the omitted parameters.

Even if you specify that the return code be placed in the parameter list, it is also placed in register 15 to accommodate high-level languages that support special return code processing.

CONNECT: Syntax and usage

CONNECT initializes a connection to DB2. You should not confuse the CONNECT function of the call attachment facility with the DB2 CONNECT statement that accesses a remote location within DB2.
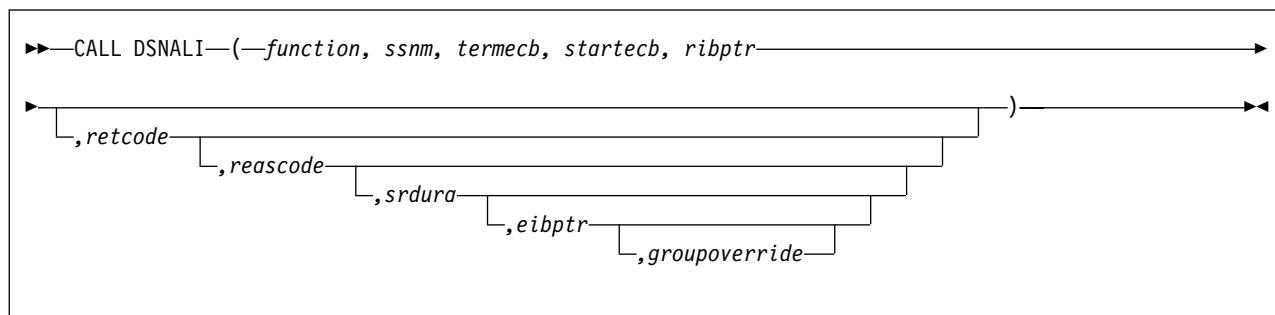
Figure 215. DSNALI connect function

Parameters point to the following areas:

function
    12-byte area containing *CONNECT* followed by five blanks.

ssnm
    4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made.

    If you specify the group attachment name, the program connects to the DB2 on the MVS system on which the program is running. When you specify a group attachment name and a start-up ECB, DB2 ignores the start-up ECB. If you need to use a start-up ECB, specify a subsystem name, rather than a group attachment name. That subsystem name must be different from the group attachment name.

    If your *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

termecb
    The application's event control block (ECB) for DB2 termination. DB2 posts this ECB when the operator enters the STOP DB2 command or when DB2 is undergoing abend. It indicates the type of termination by a POST code, as follows:

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

Before you check *termecb* in your CAF application program, first check the return code and reason code from the CONNECT call to ensure that the call completed successfully. See "Checking return codes and reason codes" on page 756 for more information.

startecb
The application's start-up ECB. If DB2 has not yet started when the application issues the call, DB2 posts the ECB when it successfully completes its startup processing. DB2 posts at most one startup ECB per address space. The ECB is the one associated with the most recent CONNECT call from that address space. Your application program must examine any nonzero CAF/DB2 reason codes *before* issuing a WAIT on this ECB.

If *ssnm* is a group attachment name, the first DB2 subsystem that starts on the local OS/390 system and matches the specified group attachment name posts the ECB.

ribptr
A 4-byte area in which CAF places the address of the release information block (RIB) after the call. You can determine what release level of DB2 you are currently running by examining field RIBREL. You can determine the modification level within the release level by examining fields RIBCNUMB and RIBCINFO. If the value in RIBCNUMB is greater than zero, check RIBCINFO for modification levels.

If the RIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *ribptr* points is below the 16-megabyte line.

Your program does not have to use the release information block, but it cannot omit the *ribptr* parameter.

Macro DSNDRIB maps the release information block (RIB). It can be found in *prefix*.SDSNMACS(DSNDRIB).

retcode
A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

srdura
A 10-byte area containing the string 'SRDURA(CD)'. This field is optional. If it is provided, the value in the CURRENT DEGREE special register stays in effect from CONNECT until DISCONNECT. If it is not provided, the value in the CURRENT DEGREE special register stays in effect from OPEN until CLOSE. If you specify this parameter in any language except assembler, you must also

specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

eibptr

A 4-byte area in which CAF puts the address of the environment information block (EIB). The EIB contains information that you can use if you are connecting to a DB2 subsystem that is part of a data sharing group. For example, you can determine the name of the data sharing group and member to which you are connecting. If the DB2 subsystem that you connect to is not part of a data sharing group, then the fields in the EIB that are related to data sharing are blank. If the EIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *eibptr* points is below the 16-megabyte line.

You can omit this parameter when you make a CONNECT call.

If you specify this parameter in any language except assembler, you must also specify the return code, reason code, and *srdura* parameters. In assembler language, you can omit the return code, reason code, and *srdura* parameters by specifying commas as place-holders.

Macro DSNDEIB maps the EIB. It can be found in *prefix*.SDSNMACS(DSNDEIB).

groupoverride

An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code, reason code, *srdura*, and *eibptr* parameters. In assembler language, you can omit the return code, reason code, *srdura*, and *eibptr* parameters by specifying commas as place-holders.

Usage: CONNECT establishes the caller's task as a user of DB2 services. If no other task in the address space currently holds a connection with the subsystem named by *ssnm*, then CONNECT also initializes the address space for communication to the DB2 address spaces. CONNECT establishes the address space's cross memory authorization to DB2 and builds address space control blocks.

In a data sharing environment, use the *groupoverride* parameter on a CONNECT call when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

Using a CONNECT call is optional. The first request from a task, either OPEN, or an SQL or IFI call, causes CAF to issue an implicit CONNECT request. If a task is connected implicitly, the connection to DB2 is terminated either when you execute CLOSE or when the task terminates.

Establishing task and address space level connections is essentially an initialization function and involves significant overhead. If you use CONNECT to establish a task connection explicitly, it terminates when you use DISCONNECT or when the task

terminates. The explicit connection minimizes the overhead by ensuring that the connection to DB2 remains after CLOSE deallocates a plan.

You can run CONNECT from any or all tasks in the address space, but the address space level is initialized only once when the first task connects.

If a task does not issue an explicit CONNECT or OPEN, the implicit connection from the first SQL or IFI call specifies a default DB2 subsystem name. A systems programmer or administrator determines the default subsystem name when installing DB2. Be certain that you know what the default name is and that it names the specific DB2 subsystem you want to use.

Practically speaking, you must not mix explicit CONNECT and OPEN requests with implicitly established connections in the same address space. Either explicitly specify which DB2 subsystem you want to use or allow all requests to use the default subsystem.

Use CONNECT when:
- You need to specify a particular (non-default) subsystem name (*ssnm*).
- You need the value of the CURRENT DEGREE special register to last as long as the connection (*srdura*).
- You need to monitor the DB2 start-up ECB (*startecb*), the DB2 termination ECB (*termecb*), or the DB2 release level.
- Multiple tasks in the address space will be opening and closing plans.
- A single task in the address space will be opening and closing plans more than once.

The other parameters of CONNECT enable the caller to learn:
- That the operator has issued a STOP DB2 command. When this happens, DB2 posts the termination ECB, *termecb*. Your application can either wait on or just look at the ECB.
- That DB2 is undergoing abend. When this happens, DB2 posts the termination ECB, *termecb*.
- That DB2 is once again available (after a connection attempt that failed because DB2 was down). Wait on or look at the start-up ECB, *startecb*. DB2 ignores this ECB if it was active at the time of the CONNECT request, or if the CONNECT request was to a group attachment name.
- The current release level of DB2. Access the RIBREL field in the release information block (RIB).

Do not issue CONNECT requests from a TCB that already has an active DB2 connection. (See "Summary of CAF behavior" on page 749 and "Error messages and dsntrace" on page 752 for more information on CAF errors.)

Table 82 shows a CONNECT call in each language.

Table 82. Examples of CAF CONNECT calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,TERMECB,STARTECB, RIBPTR,RETCODE,REASCODE,SRDURA,EIBPTR, GRPOVER)
C	fnret=dsnali(&functn[0],&ssid[0], &tecb, &secb,&ribptr,&retcode, &reascode, &srdura[0], &eibptr, &grpover[0]);

Table 82. Examples of CAF CONNECT calls (continued)

Language	Call example
COBOL	CALL 'DSNALI' USING FUNCTN SSID TERMECB STARTECB RIBPTR RETCODE REASCODE SRDURA EIBPTR GRPOVER.
FORTRAN	CALL DSNALI(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR, RETCODE,REASCODE,SRDURA,EIBPTR,GRPOVER)
PL/I	CALL DSNALI(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE, REASCODE,SRDURA,EIBPTR,GRPOVER);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
                  int DSNALI(
                    char * functn,
                      ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

OPEN: Syntax and usage

OPEN allocates resources to run the specified plan. Optionally, OPEN requests a DB2 connection for the issuing task.
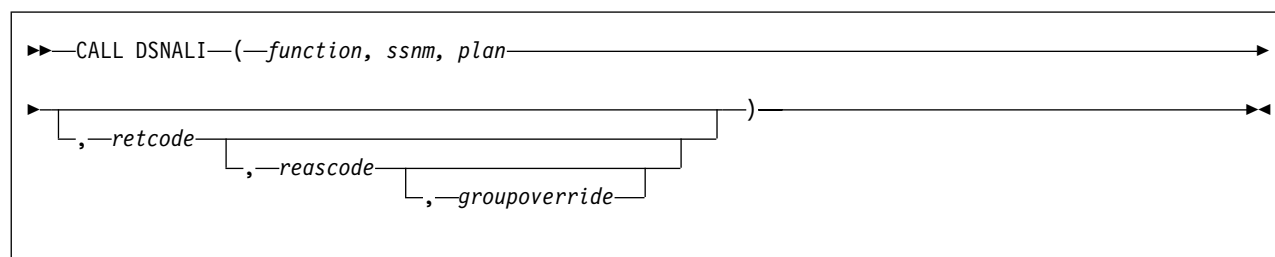
Figure 216. DSNALI OPEN function

Parameters point to the following areas:

function
A 12-byte area containing the word OPEN followed by eight blanks.

ssnm
A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group). Optionally, OPEN establishes a connection from *ssnm* to the named DB2 subsystem. If your *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

plan
An 8-byte DB2 plan name.

retcode
A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

groupoverride
An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage: OPEN allocates DB2 resources needed to run the plan or issue IFI requests. If the requesting task does not already have a connection to the named DB2 subsystem, then OPEN establishes it.

OPEN allocates the plan to the DB2 subsystem named in *ssnm*. The *ssnm* parameter, like the others, is required, even if the task issues a CONNECT call. If a task issues CONNECT followed by OPEN, then the subsystem names for both calls must be the same.

In a data sharing environment, use the *groupoverride* parameter on an OPEN call when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

The use of OPEN is optional. If you do not use OPEN, the action of OPEN occurs on the first SQL or IFI call from the task, using the defaults listed under "Implicit connections" on page 734.

Do not use OPEN if the task already has a plan allocated.

Table 83 shows an OPEN call in each language.

Table 83. Examples of CAF OPEN calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)
C	fnret=dsnali(&functn[0],&ssid[0], &planname[0],&retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNALI' USING FUNCTN SSID PLANNAME RETCODE REASCODE GRPOVER.
FORTRAN	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
                int DSNALI(
                  char * functn,
                    ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

CLOSE: Syntax and usage

CLOSE deallocates the plan and optionally disconnects the task, and possibly the address space, from DB2.

```
►►──CALL DSNALI──(──function, termop─────────────────────────)────────────►◄
                                  └─,──retcode─┐
                                       └─,──reascode─┘
```
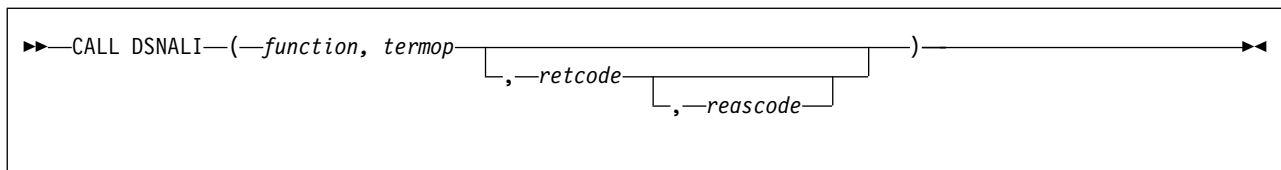
Figure 217. DSNALI CLOSE function

Parameters point to the following areas:

function
    A 12-byte area containing the word CLOSE followed by seven blanks.

termop
    A 4-byte terminate option, with one of these values:

    SYNC   Commit any modified data

    ABRT   Roll back data to the previous commit point.

retcode
    A 4-byte area in which CAF should place the return code.

    This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
    A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

    This field is optional. If specified, you must also specify *retcode*.

Usage: CLOSE deallocates the created plan either explicitly using OPEN or implicitly at the first SQL call.

If you did not issue a CONNECT for the task, CLOSE also deletes the task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross memory authorization.

Do not use CLOSE when your current task does not have a plan allocated.

Using CLOSE is optional. If you omit it, DB2 performs the same actions when your task terminates, using the SYNC parameter if termination is normal and the ABRT parameter if termination is abnormal. (The function is an implicit CLOSE.) If the objective is to shut down your application, you can improve shut down performance by using CLOSE explicitly before the task terminates.

If you want to use a new plan, you must issue an explicit CLOSE, followed by an OPEN, specifying the new plan name.

If DB2 terminates, a task that did not issue CONNECT should explicitly issue CLOSE, so that CAF can reset its control blocks to allow for future connections. This CLOSE returns the *reset accomplished* return code (+004) and reason code X'00C10824'. If you omit CLOSE, then when DB2 is back on line, the task's next connection request fails. You get either the message *Your TCB does not have a*

connection, with X'00F30018' in register 0, or CAF error message DSNA201I or DSNA202I, depending on what your application tried to do. The task must then issue CLOSE before it can reconnect to DB2.

A task that issued CONNECT explicitly should issue DISCONNECT to cause CAF to reset its control blocks when DB2 terminates. In this case, CLOSE is not necessary.

Table 84 shows a CLOSE call in each language.

Table 84. Examples of CAF CLOSE calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,TERMOP,RETCODE, REASCODE)
C	fnret=dsnali(&functn[0], &termop[0], &retcode,&reascode);
COBOL	CALL 'DSNALI' USING FUNCTN TERMOP RETCODE REASCODE.
FORTRAN	CALL DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE)
PL/I	CALL DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
            int DSNALI(
           char * functn,
          ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

DISCONNECT: Syntax and usage

DISCONNECT terminates a connection to DB2.
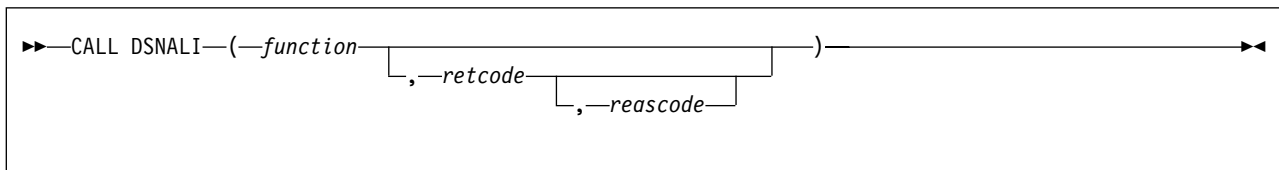
Figure 218. DSNALI DISCONNECT function

The single parameter points to the following area:

function
    A 12-byte area containing the word DISCONNECT followed by two blanks.

retcode
    A 4-byte area in which CAF places the return code.

    This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
    A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

    This field is optional. If specified, you must also specify *retcode*.

Usage: DISCONNECT removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross memory authorization.

Only those tasks that issued CONNECT explicitly can issue DISCONNECT. If CONNECT was not used, then DISCONNECT causes an error.

If an OPEN is in effect when the DISCONNECT is issued (that is, a plan is allocated), CAF issues an implicit CLOSE with the SYNC parameter.

Using DISCONNECT is optional. Without it, DB2 performs the same functions when the task terminates. (The function is an implicit DISCONNECT.) If the objective is to shut down your application, you can improve shut down performance if you request DISCONNECT explicitly before the task terminates.

If DB2 terminates, a task that issued CONNECT must issue DISCONNECT to reset the CAF control blocks. The function returns the *reset accomplished* return codes and reason codes (+004 and X'00C10824'), and ensures that future connection requests from the task work when DB2 is back on line.

A task that did not issue CONNECT explicitly must issue CLOSE to reset the CAF control blocks when DB2 terminates.

Table 85 shows a DISCONNECT call in each language.

Table 85. Examples of CAF DISCONNECT calls

Language	Call example
Assembler	CALL DSNALI(,FUNCTN,RETCODE,REASCODE)
C	fnret=dsnali(&functn[0], &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN RETCODE REASCODE.
FORTRAN	CALL DSNALI(FUNCTN,RETCODE,REASCODE)
PL/I	CALL DSNALI(FUNCTN,RETCODE,REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C #pragma linkage(dsnali, OS)

C⁺⁺ extern "OS" {
                int DSNALI(
                   char * functn,
                      ...); }

PL/I DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

TRANSLATE: Syntax and usage

You can use TRANSLATE to convert a DB2 hexadecimal error reason code into a signed integer SQLCODE and a printable error message text. The SQLCODE and message text appear in the caller's SQLCA. You cannot call the TRANSLATE function from the FORTRAN language.

TRANSLATE is useful only after an OPEN fails, and then only if you used an explicit CONNECT before the OPEN request. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

```
►►──CALL DSNALI──(─function, sqlca──┬──────────────────────────┬──)──────────────►◄
                                     └─,─retcode─┬────────────┬─┘
                                                 └─,─reascode─┘
```
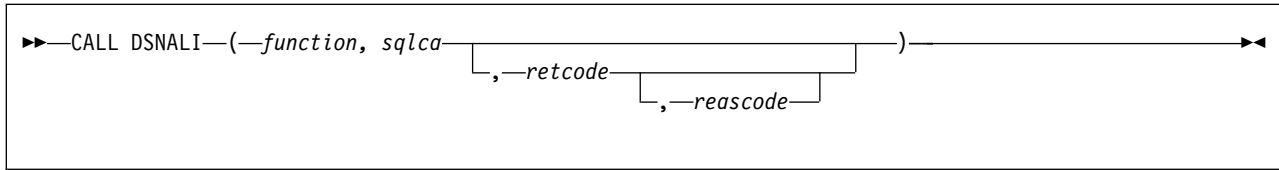
Figure 219. DSNALI TRANSLATE function

Parameters point to the following areas:

function
> A 12-byte area containing the word TRANSLATE followed by three blanks.

sqlca
> The program's SQL communication area (SQLCA).

retcode
> A 4-byte area in which CAF places the return code.
>
> This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
> A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.
>
> This field is optional. If specified, you must also specify *retcode*.

Usage: Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that CAF returns in register 0 following an OPEN service request. DB2 places the information into the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function can translate those codes beginning with X'00F3', but it does not translate CAF reason codes beginning with X'00C1'. If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original DB2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails; in which case, register 0 is set to X'C10205' and register 15 to 200.

Table 86 shows a TRANSLATE call in each language.

Table 86. Examples of CAF TRANSLATE calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SQLCA,RETCODE, REASCODE)
C	fnret=dsnali(&functn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN SQLCA RETCODE REASCODE.
PL/I	CALL DSNALI(FUNCTN,SQLCA,RETCODE, REASCODE);

Language	Call example

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
                int DSNALI(
                    char * functn,
                        ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

Summary of CAF behavior

Table 87 summarizes CAF behavior after various inputs from application programs. Use it to help plan the calls your program makes, and to help understand where CAF errors can occur. Careful use of this table can avoid major structural problems in your application.

In the table, an error shows as *Error nnn*. The corresponding reason code is X'00C10'*nnn*; the message number is DSNA*nnn*I or DSNA*nnn*E. For a list of reason codes, see "CAF return codes and reason codes" on page 752.

Table 87. Effects of CAF calls, as dependent on connection history

Previous function	Next function					
	CONNECT	**OPEN**	**SQL**	**CLOSE**	**DISCONNECT**	**TRANSLATE**
Empty: first call	CONNECT	OPEN	CONNECT, OPEN, followed by the SQL or IFI call	Error 203	Error 204	Error 205
CONNECT	Error 201	OPEN	OPEN, followed by the SQL or IFI call	Error 203	DISCONNECT	TRANSLATE
CONNECT followed by OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE[1]	DISCONNECT	TRANSLATE
CONNECT followed by SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE[1]	DISCONNECT	TRANSLATE
OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE[2]	Error 204	TRANSLATE
SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE[2]	Error 204	TRANSLATE[3]

Notes:

1. The task and address space connections remain active. If CLOSE fails because DB2 was down, then the CAF control blocks are reset, the function produces return code 4 and reason code XX'00C10824', and CAF is ready for more connection requests when DB2 is again on line.

2. The connection for the task is terminated. If there are no other connected tasks in the address space, the address space level connection terminates also.

3. A TRANSLATE request is accepted, but in this case it is redundant. CAF automatically issues a TRANSLATE request when an SQL or IFI request fails.

Table 87 on page 749 uses the following conventions:

- *The top row* lists the possible CAF functions that programs can use as their call.
- *The first column* lists the task's most recent history of connection requests. For example, *CONNECT followed by OPEN* means that the task issued CONNECT and then OPEN with no other CAF calls in between.
- *The intersection of a row and column* shows the effect of the next call if it follows the corresponding connection history. For example, if the call is OPEN and the connection history is CONNECT, the effect is OPEN: the OPEN function is performed. If the call is SQL and the connection history is empty (meaning that the SQL call is the first CAF function the program), the effect is that an implicit CONNECT and OPEN function is performed, followed by the SQL function.

Sample scenarios

This section shows sample scenarios for connecting tasks to DB2.

A single task with implicit connections

The simplest connection scenario is a single task making calls to DB2, using no explicit CALL DSNALI statements. The task implicitly connects to the default subsystem name, using the default plan name.

When the task terminates:

- Any database changes are committed (if termination was normal) or rolled back (if termination was abnormal).
- The active plan and all database resources are deallocated.
- The task and address space connections to DB2 are terminated.

A single task with explicit connections

A more complex scenario, but still with a single task, is this:

```
CONNECT
    OPEN        allocate a plan
    SQL or IFI call
⋮
    CLOSE       deallocate the current plan
    OPEN        allocate a new plan
    SQL or IFI call
⋮
    CLOSE
DISCONNECT
```

A task can have a connection to one and only one DB2 subsystem at any point in time. A CAF error occurs if the subsystem name on OPEN does not match the one on CONNECT. To switch to a different subsystem, the application must disconnect from the current subsystem, then issue a connect request specifying a new subsystem name.

Several tasks

In this scenario, multiple tasks within the address space are using DB2 services. Each task must explicitly specify the same subsystem name on either the CONNECT or OPEN function request. Task 1 makes no SQL or IFI calls. Its purpose is to monitor the DB2 termination and start-up ECBs, and to check the DB2 release level.

```
        TASK 1          TASK 2          TASK 3          TASK n

        CONNECT
                        OPEN            OPEN            OPEN
                        SQL             SQL             SQL
                        ...             ...             ...
                        CLOSE           CLOSE           CLOSE
                        OPEN            OPEN            OPEN
                        SQL             SQL             SQL
                        ...             ...             ...
                        CLOSE           CLOSE           CLOSE
        DISCONNECT
```

Exits from your application

You can provide exits from your application for the purposes described in the following text.

Attention exits

An attention exit enables you to regain control from DB2, during long-running or erroneous requests, by detaching the TCB currently waiting on an SQL or IFI request to complete. DB2 detects the abend caused by DETACH and performs termination processing (including ROLLBACK) for that task.

The call attachment facility has no attention exits. You can provide your own if necessary. However, DB2 uses enabled unlocked task (EUT) functional recovery routines (FRRs), so if you request attention while DB2 code is running, your routine may not get control.

Recovery routines

The call attachment facility has no abend recovery routines.

Your program can provide an abend exit routine. It must use tracking indicators to determine if an abend occurred during DB2 processing. If an abend occurs while DB2 has control, you have these choices:

- Allow task termination to complete. Do not retry the program. DB2 detects task termination and terminates the thread with the ABRT parameter. You lose all database changes back to the last SYNC or COMMIT point.

  This is the only action that you can take for abends that CANCEL or DETACH cause. You cannot use additional SQL statements at this point. If you attempt to execute another SQL statement from the application program or its recovery routine, a return code of +256 and a reason code of X'00F30083' occurs.

- In an ESTAE routine, issue CLOSE with the ABRT parameter followed by DISCONNECT. The ESTAE exit routine can retry so that you do not need to re-instate the application task.

Standard MVS functional recovery routines (FRRs) can cover only code running in service request block (SRB) mode. Because DB2 does not support calls from SRB mode routines, you can use only enabled unlocked task (EUT) FRRs in your routines that call DB2.

Do not have an EUT FRR active when using CAF, processing SQL requests, or calling IFI.

An EUT FRR can be active, but it cannot retry failing DB2 requests. An EUT FRR retry bypasses DB2's ESTAE routines. The next DB2 request of any type, including DISCONNECT, fails with a return code of +256 and a reason code of X'00F30050'.

With MVS, if you have an active EUT FRR, all DB2 requests fail, including the initial CONNECT or OPEN. The requests fail because DB2 always creates an ARR-type ESTAE, and MVS/ESA does not allow the creation of ARR-type ESTAEs when an FRR is active.

Error messages and dsntrace

CAF produces no error messages unless you allocate a DSNTRACE data set. If you allocate a DSNTRACE data set either dynamically or by including a //DSNTRACE DD statement in your JCL, CAF writes diagnostic trace message to that data set. You can refer to "Sample JCL for using CAF" on page 753 for sample JCL that allocates a DSNTRACE data set. The trace message numbers contain the last 3 digits of the reason codes.

CAF return codes and reason codes

CAF returns the return codes and reason codes either to the corresponding parameters named in a CAF call or, if you choose not to use those parameters, to registers 15 and 0. Detailed explanations of the reason codes appear in *DB2 Messages and Codes*.

When the reason code begins with X'00F3' (except for X'00F30006'), you can use the CAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, CAF returns standard SQLCODEs in the SQLCA. See Part 1 of *DB2 Messages and Codes* for a list of those return codes and their meanings. CAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

Table 88. CAF return codes and reason codes

Return code	Reason code	Explanation
0	X'00000000'	Successful completion.
4	X'00C10823'	Release level mismatch between DB2 and the and the call attachment facility code.
4	X'00C10824'	CAF reset complete. Ready to make a new connection.
200 (note 1)	X'00C10201'	Received a second CONNECT from the same TCB. The first CONNECT could have been implicit or explicit.
200 (note 1)	X'00C10202'	Received a second OPEN from the same TCB. The first OPEN could have been implicit or explicit.
200 (note 1)	X'00C10203'	CLOSE issued when there was no active OPEN.
200 (note 1)	X'00C10204'	DISCONNECT issued when there was no active CONNECT, or the AXSET macro was issued between CONNECT and DISCONNECT.
200 (note 1)	X'00C10205'	TRANSLATE issued when there was no connection to DB2.
200 (note 1)	X'00C10206'	Wrong number of parameters or the end-of-list bit was off.

Table 88. CAF return codes and reason codes (continued)

Return code	Reason code	Explanation
200 (note 1)	X'00C10207'	Unrecognized function parameter.
200 (note 1)	X'00C10208'	Received requests to access two different DB2 subsystems from the same TCB.
204	(note 2)	CAF system error. Probable error in the attach or DB2.

Notes:

1. A CAF error probably caused by errors in the parameter lists coming from application programs. CAF errors do not change the current state of your connection to DB2; you can continue processing with a corrected request.

2. System errors cause abends. For an explanation of the abend reason codes, see Part 3 of *DB2 Messages and Codes*. If tracing is on, a descriptive message is written to the DSNTRACE data set just before the abend.

Subsystem support subcomponent codes (X'00F3')

These reason codes are issued by the subsystem support for allied memories, a part of the DB2 subsystem support subcomponent that services all DB2 connection and work requests. For more information on the codes, along with abend and subsystem termination reason codes issued by other parts of subsystem support, see Part 3 of *DB2 Messages and Codes*.

Program examples

The following pages contain sample JCL and assembler programs that access the call attachment facility (CAF).

Sample JCL for using CAF

The sample JCL that follows is a model for using CAF in a batch (non-TSO) environment. The DSNTRACE statement shown in this example is optional.

```
//jobname    JOB    MVS_jobcard_information
//CAFJCL     EXEC   PGM=CAF_application_program
//STEPLIB    DD     DSN=application_load_library
//           DD     DSN=DB2_load_library
       .
       .
       .
//SYSPRINT   DD     SYSOUT=*
//DSNTRACE   DD     SYSOUT=*
//SYSUDUMP   DD     SYSOUT=*
```

Sample assembler code for using CAF

The following sections show parts of a sample assembler program using the call attachment facility. It demonstrates the basic techniques for making CAF calls but does not show the code and MVS macros needed to support those calls. For example, many applications need a two-task structure so that attention-handling routines can detach connected subtasks to regain control from DB2. This structure is not shown in the code that follows.

These code segments assume the existence of a WRITE macro. Anywhere you find this macro in the code is a good place for you to substitute code of your own. You must decide what you want your application to do in those situations; you probably do not want to write the error messages shown.

Loading and deleting the CAF language interface

The following code segment shows how an application can load entry points DSNALI and DSNHLI2 for the call attachment language interface. Storing the entry points in variables LIALI and LISQL ensures that the application has to load the entry points only once.

When the module is done with DB2, you should delete the entries.

```
***************************** GET LANGUAGE INTERFACE ENTRY ADDRESSES
        LOAD  EP=DSNALI          Load the CAF service request EP
        ST    R0,LIALI           Save this for CAF service requests
        LOAD  EP=DSNHLI2         Load the CAF SQL call Entry Point
        ST    R0,LISQL           Save this for SQL calls
*       .
*       .     Insert connection service requests and SQL calls here
*       .
        DELETE EP=DSNALI         Correctly maintain use count
        DELETE EP=DSNHLI2        Correctly maintain use count
```

Establishing the connection to DB2

Figure 220 on page 755 shows how to issue explicit requests for certain actions (CONNECT, OPEN, CLOSE, DISCONNECT, and TRANSLATE), using the CHEKCODE subroutine to check the return reason codes from CAF:

```
****************************** CONNECT ******************************
        L     R15,LIALI          Get the Language Interface address
        MVC   FUNCTN,CONNECT     Get the function to call
        CALL  (15),(FUNCTN,SSID,TECB,SECB,RIBPTR),VL,MF=(E,CAFCALL)
        BAL   R14,CHEKCODE       Check the return and reason codes
        CLC   CONTROL,CONTINUE   Is everything still OK
        BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
        USING R8,RIB             Prepare to access the RIB
        L     R8,RIBPTR          Access RIB to get DB2 release level
        WRITE 'The current DB2 release level is' RIBREL

****************************** OPEN ******************************
        L     R15,LIALI          Get the Language Interface address
        MVC   FUNCTN,OPEN        Get the function to call
        CALL  (15),(FUNCTN,SSID,PLAN),VL,MF=(E,CAFCALL)
        BAL   R14,CHEKCODE       Check the return and reason codes

****************************** SQL ******************************
*             Insert your SQL calls here.  The DB2 Precompiler
*             generates calls to entry point DSNHLI.  You should
*             specify the precompiler option ATTACH(CAF), or code
*             a dummy entry point named DSNHLI to intercept
*             all SQL calls.  A dummy DSNHLI is shown below.
****************************** CLOSE ******************************
        CLC   CONTROL,CONTINUE   Is everything still OK?
        BNE   EXIT               If CONTROL not 'CONTINUE', shut down
        MVC   TRMOP,ABRT         Assume termination with ABRT parameter
        L     R4,SQLCODE         Put the SQLCODE into a register
        C     R4,CODE0           Examine the SQLCODE
        BZ    SYNCTERM           If zero, then CLOSE with SYNC parameter
        C     R4,CODE100         See if SQLCODE was 100
        BNE   DISC               If not 100, CLOSE with ABRT parameter
SYNCTERM MVC  TRMOP,SYNC         Good code, terminate with SYNC parameter
DISC    DS    0H                 Now build the CAF parmlist
        L     R15,LIALI          Get the Language Interface address
        MVC   FUNCTN,CLOSE       Get the function to call
        CALL  (15),(FUNCTN,TRMOP),VL,MF=(E,CAFCALL)
        BAL   R14,CHEKCODE       Check the return and reason codes

****************************** DISCONNECT ******************************
        CLC   CONTROL,CONTINUE   Is everything still OK
        BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
        L     R15,LIALI          Get the Language Interface address
        MVC   FUNCTN,DISCON      Get the function to call
        CALL  (15),(FUNCTN),VL,MF=(E,CAFCALL)
        BAL   R14,CHEKCODE       Check the return and reason codes
```

Figure 220. CHEKCODE Subroutine for connecting to DB2

The code does not show a task that waits on the DB2 termination ECB. If you like, you can code such a task and use the MVS WAIT macro to monitor the ECB. You probably want this task to detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. This sample waits on the startup ECB at its own task level.

On entry, the code assumes that certain variables are already set:

Variable	Usage
LIALI	The entry point that handles DB2 connection service requests.
LISQL	The entry point that handles SQL calls.
SSID	The DB2 subsystem identifier.
TECB	The address of the DB2 termination ECB.

SECB	The address of the DB2 start-up ECB.
RIBPTR	A fullword that CAF sets to contain the RIB address.
PLAN	The plan name to use on the OPEN call.
CONTROL	Used to shut down processing because of unsatisfactory return or reason codes. Subroutine CHEKCODE sets CONTROL.
CAFCALL	List-form parameter area for the CALL macro.

Checking return codes and reason codes

Figure 221 on page 757 illustrates a way to check the return codes and the DB2 termination ECB after each connection service request and SQL call. The routine sets the variable CONTROL to control further processing within the module.

```
    *********************************************************************
    * CHEKCODE PSEUDOCODE                                               *
    *********************************************************************
    *IF TECB is POSTed with the ABTERM or FORCE codes
    *  THEN
    *    CONTROL = 'SHUTDOWN'
    *    WRITE 'DB2 found FORCE or ABTERM, shutting down'
    *  ELSE                              /* Termination ECB was not POSTed  */
    *    SELECT (RETCODE)                /* Look at the return code         */
    *      WHEN (0) ;                    /* Do nothing; everything is OK    */
    *      WHEN (4) ;                    /* Warning                         */
    *        SELECT (REASCODE)           /* Look at the reason code         */
    *          WHEN ('00C10823'X)        /* DB2 / CAF release level mismatch*/
    *            WRITE 'Found a mismatch between DB2 and CAF release levels'
    *          WHEN ('00C10824'X)        /* Ready for another CAF call      */
    *            CONTROL = 'RESTART'     /* Start over, from the top        */
    *          OTHERWISE
    *            WRITE 'Found unexpected R0 when R15 was 4'
    *            CONTROL = 'SHUTDOWN'
    *        END INNER-SELECT
    *      WHEN (8,12)                   /* Connection failure              */
    *        SELECT (REASCODE)           /* Look at the reason code         */
    *          WHEN ('00F30002'X,        /* These mean that DB2 is down but */
    *                '00F30012'X)        /* will POST SECB when up again    */
    *            DO
    *              WRITE 'DB2 is unavailable. I'll tell you when it's up.'
    *              WAIT SECB             /* Wait for DB2 to come up          */
    *              WRITE 'DB2 is now available.'
    *            END
    *          /*********************************************************/
    *          /* Insert tests for other DB2 connection failures here.  */
    *          /* CAF Externals Specification lists other codes you can */
    *          /* receive.  Handle them in whatever way is appropriate  */
    *          /* for your application.                                 */
    *          /*********************************************************/
    *          OTHERWISE                 /* Found a code we're not ready for*/
    *            WRITE 'Warning: DB2 connection failure. Cause unknown'
    *            CALL DSNALI ('TRANSLATE',SQLCA) /* Fill in SQLCA        */
    *            WRITE SQLCODE and SQLERRM
    *        END INNER-SELECT
    *      WHEN (200)
    *        WRITE 'CAF found user error. See DSNTRACE dataset'
    *      WHEN (204)
    *        WRITE 'CAF system error. See DSNTRACE data set'
    *      OTHERWISE
    *        CONTROL = 'SHUTDOWN'
    *        WRITE 'Got an unrecognized return code'
    *    END MAIN SELECT
    *    IF (RETCODE > 4) THEN           /* Was there a connection problem?*/
    *      CONTROL = 'SHUTDOWN'
    * END CHEKCODE
```

Figure 221. Subroutine to check return codes from CAF and DB2, in assembler (Part 1 of 3)

```
     ***********************************************************************
     * Subroutine CHEKCODE checks return codes from DB2 and Call Attach.
     * When CHEKCODE receives control, R13 should point to the caller's
     * save area.
     ***********************************************************************
CHEKCODE DS    0H
         STM   R14,R12,12(R13)    Prolog
         ST    R15,RETCODE        Save the return code
         ST    R0,REASCODE        Save the reason code
         LA    R15,SAVEAREA       Get save area address
         ST    R13,4(,R15)        Chain the save areas
         ST    R15,8(,R13)        Chain the save areas
         LR    R13,R15            Put save area address in R13
*        ******************** HUNT FOR FORCE OR ABTERM ***************
         TM    TECB,POSTBIT       See if TECB was POSTed
         BZ    DOCHECKS           Branch if TECB was not POSTed
         CLC   TECBCODE(3),QUIESCE   Is this "STOP DB2 MODE=FORCE"
         BE    DOCHECKS           If not QUIESCE, was FORCE or ABTERM
         MVC   CONTROL,SHUTDOWN   Shutdown
         WRITE 'Found found FORCE or ABTERM, shutting down'
         B     ENDCCODE           Go to the end of CHEKCODE
DOCHECKS DS    0H                 Examine RETCODE and REASCODE
*        ******************** HUNT FOR 0 ****************************
         CLC   RETCODE,ZERO       Was it a zero?
         BE    ENDCCODE           Nothing to do in CHEKCODE for zero
*        ******************** HUNT FOR 4 ****************************
         CLC   RETCODE,FOUR       Was it a 4?
         BNE   HUNT8              If not a 4, hunt eights
         CLC   REASCODE,C10823    Was it a release level mismatch?
         BNE   HUNT824            Branch if not an 823
         WRITE 'Found a mismatch between DB2 and CAF release levels'
         B     ENDCCODE           We are done. Go to end of CHEKCODE
HUNT824  DS    0H                 Now look for 'CAF reset' reason code
         CLC   REASCODE,C10824    Was it 4? Are we ready to restart?
         BNE   UNRECOG            If not 824, got unknown code
         WRITE 'CAF is now ready for more input'
         MVC   CONTROL,RESTART    Indicate that we should re-CONNECT
         B     ENDCCODE           We are done. Go to end of CHEKCODE
UNRECOG  DS    0H
         WRITE 'Got RETCODE = 4 and an unrecognized reason code'
         MVC   CONTROL,SHUTDOWN   Shutdown, serious problem
         B     ENDCCODE           We are done. Go to end of CHEKCODE
*        ******************** HUNT FOR 8 ****************************
HUNT8    DS    0H
         CLC   RETCODE,EIGHT      Hunt return code of 8
         BE    GOT8OR12
         CLC   RETCODE,TWELVE     Hunt return code of 12
         BNE   HUNT200
GOT8OR12 DS    0H                 Found return code of 8 or 12
         WRITE 'Found RETCODE of 8 or 12'
         CLC   REASCODE,F30002    Hunt for X'00F30002'
         BE    DB2DOWN
```

Figure 221. Subroutine to check return codes from CAF and DB2, in assembler (Part 2 of 3)

```
              CLC   REASCODE,F30012    Hunt for X'00F30012'
              BE    DB2DOWN
              WRITE 'DB2 connection failure with an unrecognized REASCODE'
              CLC   SQLCODE,ZERO       See if we need TRANSLATE
              BNE   A4TRANS            If not blank, skip TRANSLATE
*             ******************** TRANSLATE unrecognized RETCODEs ********
              WRITE 'SQLCODE 0 but R15 not, so TRANSLATE to get SQLCODE'
              L     R15,LIALI          Get the Language Interface address
              CALL  (15),(TRANSLAT,SQLCA),VL,MF=(E,CAFCALL)
              C     R0,C10205          Did the TRANSLATE work?
              BNE   A4TRANS            If not C10205, SQLERRM now filled in
              WRITE 'Not able to TRANSLATE the connection failure'
              B     ENDCCODE           Go to end of CHEKCODE
A4TRANS  DS   0H                       SQLERRM must be filled in to get here
*        Note: your code should probably remove the X'FF'
*        separators and format the SQLERRM feedback area.
*        Alternatively, use DB2 Sample Application DSNTIAR
*        to format a message.
              WRITE 'SQLERRM is:' SQLERRM
              B     ENDCCODE           We are done. Go to end of CHEKCODE
DB2DOWN  DS   0H                       Hunt return code of 200
              WRITE 'DB2 is down and I will tell you when it comes up'
              WAIT  ECB=SECB           Wait for DB2 to come up
              WRITE 'DB2 is now available'
              MVC   CONTROL,RESTART    Indicate that we should re-CONNECT
              B     ENDCCODE
*             ******************** HUNT FOR 200 ***************************
HUNT200  DS   0H                       Hunt return code of 200
              CLC   RETCODE,NUM200     Hunt 200
              BNE   HUNT204
              WRITE 'CAF found user error, see DSNTRACE data set'
              B     ENDCCODE           We are done. Go to end of CHEKCODE
*             ******************** HUNT FOR 204 ***************************
HUNT204  DS   0H                       Hunt return code of 204
              CLC   RETCODE,NUM204     Hunt 204
              BNE   WASSAT             If not 204, got strange code
              WRITE 'CAF found system error, see DSNTRACE data set'
              B     ENDCCODE           We are done. Go to end of CHEKCODE
*             ******************** UNRECOGNIZED RETCODE ******************
WASSAT   DS   0H
              WRITE 'Got an unrecognized RETCODE'
              MVC   CONTROL,SHUTDOWN   Shutdown
              BE    ENDCCODE           We are done. Go to end of CHEKCODE
ENDCCODE DS   0H                       Should we shut down?
              L     R4,RETCODE         Get a copy of the RETCODE
              C     R4,FOUR            Have a look at the RETCODE
              BNH   BYEBYE             If RETCODE <= 4 then leave CHEKCODE
              MVC   CONTROL,SHUTDOWN   Shutdown
BYEBYE   DS   0H                       Wrap up and leave CHEKCODE
              L     R13,4(,R13)        Point to caller's save area
              RETURN (14,12)           Return to the caller
```

Figure 221. Subroutine to check return codes from CAF and DB2, in assembler (Part 3 of 3)

Using dummy entry point DSNHLI

Each of the four DB2 attachment facilities contains an entry point named DSNHLI.
When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL
statements result in BALR instructions to DSNHLI in your program. To find the
correct DSNHLI entry point without including DSNALI in your load module, code a
subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in
the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in

DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the the difference.

In the example that follows, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
**********************************************************************
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
**********************************************************************
        DS    0D
DSNHLI  CSECT                     Begin CSECT
        STM   R14,R12,12(R13)     Prologue
        LA    R15,SAVEHLI         Get save area address
        ST    R13,4(,R15)         Chain the save areas
        ST    R15,8(,R13)         Chain the save areas
        LR    R13,R15             Put save area address in R13
        L     R15,LISQL           Get the address of real DSNHLI
        BASSM R14,R15             Branch to DSNALI to do an SQL call
*                                 DSNALI is in 31-bit mode, so use
*                                 BASSM to assure that the addressing
*                                 mode is preserved.
        L     R13,4(,R13)         Restore R13 (caller's save area addr)
        L     R14,12(,R13)        Restore R14 (return address)
        RETURN (1,12)             Restore R1-12, NOT R0 and R15 (codes)
```

Variable declarations

Figure 222 on page 761 shows declarations for some of the variables used in the previous subroutines.

```
***************************** VARIABLES *****************************
SECB     DS    F              DB2 Start-up ECB
TECB     DS    F              DB2 Termination ECB
LIALI    DS    F              DSNALI Entry Point address
LISQL    DS    F              DSNHLI2 Entry Point address
SSID     DS    CL4            DB2 Subsystem ID.  CONNECT parameter
PLAN     DS    CL8            DB2 Plan name.  OPEN parameter
TRMOP    DS    CL4            CLOSE termination option (SYNC|ABRT)
FUNCTN   DS    CL12           CAF function to be called
RIBPTR   DS    F              DB2 puts Release Info Block addr here
RETCODE  DS    F              Chekcode saves R15 here
REASCODE DS    F              Chekcode saves R0 here
CONTROL  DS    CL8            GO, SHUTDOWN, or RESTART
SAVEAREA DS    18F            Save area for CHEKCODE
***************************** CONSTANTS *****************************
SHUTDOWN DC    CL8'SHUTDOWN'   CONTROL value: Shutdown execution
RESTART  DC    CL8'RESTART '   CONTROL value: Restart execution
CONTINUE DC    CL8'CONTINUE'   CONTROL value: Everything OK, cont
CODE0    DC    F'0'            SQLCODE of 0
CODE100  DC    F'100'          SQLCODE of 100
QUIESCE  DC    XL3'000008'     TECB postcode: STOP DB2 MODE=QUIESCE
CONNECT  DC    CL12'CONNECT    ' Name of a CAF service. Must be CL12!
OPEN     DC    CL12'OPEN       ' Name of a CAF service. Must be CL12!
CLOSE    DC    CL12'CLOSE      ' Name of a CAF service. Must be CL12!
DISCON   DC    CL12'DISCONNECT ' Name of a CAF service. Must be CL12!
TRANSLAT DC    CL12'TRANSLATE  ' Name of a CAF service. Must be CL12!
SYNC     DC    CL4'SYNC'       Termination option (COMMIT)
ABRT     DC    CL4'ABRT'       Termination option (ROLLBACK)
***************************** RETURN CODES (R15) FROM CALL ATTACH ****
ZERO     DC    F'0'            0
FOUR     DC    F'4'            4
EIGHT    DC    F'8'            8
TWELVE   DC    F'12'           12  (Call Attach return code in R15)
NUM200   DC    F'200'          200 (User error)
NUM204   DC    F'204'          204 (Call Attach system error)
***************************** REASON CODES (R00) FROM CALL ATTACH ****
C10205   DC    XL4'00C10205'   Call attach could not TRANSLATE
C10823   DC    XL4'00C10823'   Call attach found a release mismatch
C10824   DC    XL4'00C10824'   Call attach ready for more input
F30002   DC    XL4'00F30002'   DB2 subsystem not up
F30011   DC    XL4'00F30011'   DB2 subsystem not up
F30012   DC    XL4'00F30012'   DB2 subsystem not up
F30025   DC    XL4'00F30025'   DB2 is stopping (REASCODE)
*
*        Insert more codes here as necessary for your application
*
***************************** SQLCA and RIB *************************
    EXEC SQL INCLUDE SQLCA
         DSNDRIB                Get the DB2 Release Information Block
***************************** CALL macro parm list *******************
CAFCALL  CALL  ,(*,*,*,*,*,*,*,*,*),VL,MF=L
```

Figure 222. Declarations for variables used in the previous subroutines

Chapter 30. Programming for the Recoverable Resource Manager Services attachment facility (RRSAF)

An application program can use the Recoverable Resource Manager Services attachment facility (RRSAF) to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. Programs that run in MVS batch, TSO foreground, and TSO background can use RRSAF.

RRSAF uses OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS). With RRSAF, you can coordinate DB2 updates with updates made by all other resource managers that also use OS/390 RRS in an MVS system.

Prerequisite knowledge: Before you consider using RRSAF, you must be familiar with the following MVS topics:
- The CALL macro and standard module linkage conventions
- Program addressing and residency options (AMODE and RMODE)
- Creating and controlling tasks; multitasking
- Functional recovery facilities such as ESTAE, ESTAI, and FRRs
- Synchronization techniques such as WAIT/POST.
- OS/390 RRS functions, such as SRRCMIT and SRRBACK.

RRSAF capabilities and restrictions

To decide whether to use RRSAF, consider the following capabilities and restrictions.

Capabilities of RRSAF applications

An application program using RRSAF can:
- Use the MVS System Authorization Facility and an external security product, such as RACF, to sign on to DB2 with the authorization ID of an end user.
- Sign on to DB2 using a new authorization ID and an existing connection and plan.
- Access DB2 from multiple MVS tasks in an address space.
- Switch a DB2 thread among MVS tasks within a single address space.
- Access the DB2 IFI.
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor (or of any DB2 code).
- Run above or below the 16MB line.
- Establish an *explicit* connection to DB2, through a call interface, with control over the exact state of the connection.
- Supply event control blocks (ECBs), for DB2 to post, that signal start-up or termination.
- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages as desired.

Task capabilities
Any task in an address space can establish a connection to DB2 through RRSAF.

Number of connections to DB2: Each task control block (TCB) can have only one connection to DB2. A DB2 service request issued by a program that runs under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Consider this when you write your application program.

Specifying a plan for a task: Each connected task can run a plan. Tasks within a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without completely breaking its connection to DB2.

Providing attention processing exits and recovery routines: RRSAF does not generate task structures, and it does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines only.

Programming language
You can write RRSAF applications in assembler language, C, COBOL, FORTRAN, and PL/I. When choosing a language to code your application in, consider these restrictions:
- If you use MVS macros (ATTACH, WAIT, POST, and so on), you must choose a programming language that supports them.
- The RRSAF TRANSLATE function is not available from FORTRAN. To use the function, code it in a routine written in another language, and then call that routine from FORTRAN.

Tracing facility
A tracing facility provides diagnostic messages that help you debug programs and diagnose errors in the RRSAF code. The trace information is available only in a SYSABEND or SYSUDUMP dump.

Program preparation
Preparing your application program to run in RRSAF is similar to preparing it to run in other environments, such as CICS, IMS, and TSO. You can prepare an RRSAF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST. For examples and guidance in program preparation, see "Chapter 20. Preparing an application program to run" on page 395.

RRSAF requirements
When you write an application to use RRSAF, be aware of the following characteristics.

Program size
The RRSAF code requires about 10K of virtual storage per address space and an additional 10KB for each TCB that uses RRSAF.

Use of LOAD
RRSAF uses MVS SVC LOAD to load a module as part of the initialization following your first service request. The module is loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces

the LOAD SVC, then you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard MVS LOAD macro.

Commit and rollback operations

To commit work in RRSAF applications, use the CPIC SRRCMIT function or the DB2 COMMIT statement. To roll back work, use the CPIC SRRBACK function or the DB2 ROLLBACK statement. For information on coding the SRRCMIT and SRRBACK functions, see *OS/390 MVS Programming: Callable Services for High-Level Languages*.

Follow these guidelines for choosing the DB2 statements or the CPIC functions for commit and rollback operations:

- Use DB2 COMMIT and ROLLBACK statements when you know that the following conditions are true:
  - The only recoverable resource accessed by your application is DB2 data managed by a single DB2 instance.
  - The address space from which syncpoint processing is initiated is the same as the address space that is connected to DB2.
- If your application accesses other recoverable resources, or syncpoint processing and DB2 access are initiated from different address spaces, use SRRCMIT and SRRBACK.

Run environment

Applications that request DB2 services must adhere to several run environment requirements. Those requirements must be met regardless of the attachment facility you use. They are not unique to RRSAF.

- The application must be running in TCB mode.
- No EUT FRRs can be active when the application requests DB2 services. If an EUT FRR is active, DB2's functional recovery can fail, and your application can receive unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. For example:
  - An application should not use RRSAF in CICS or IMS address spaces.
  - An application running in an address space that has a CAF connection to DB2 cannot connect to DB2 using RRSAF.
  - An application running in an address space that has an RRSAF connection to DB2 cannot connect to DB2 using CAF.
- One attachment facility cannot start another. This means your RRSAF application cannot use DSN, and a DSN RUN subcommand cannot call your RRSAF application.
- The language interface module for RRSAF, DSNRLI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load RRSAF below the 16MB line, you must link-edit DSNRLI again.

How to use RRSAF

To use RRSAF, you must first make available the RRSAF language interface load module, DSNRLI. For information on loading or link-editing this module, see "Accessing the RRSAF language interface" on page 766.

Your program uses RRSAF by issuing CALL DSNRLI statements with the appropriate options. For the general form of the statements, see "RRSAF function descriptions" on page 770.

The first element of each option list is a *function*, which describes the action you want RRSAF to take. For a list of available functions and what they do, see "Summary of connection functions" on page 769. The effect of any function depends in part on what functions the program has already performed. Before using any function, be sure to read the description of its usage. Also read "Summary of connection functions" on page 769, which describes the influence of previously invoked functions.

Accessing the RRSAF language interface

Figure 223 on page 767 shows the general structure of RRSAF and a program that uses it.
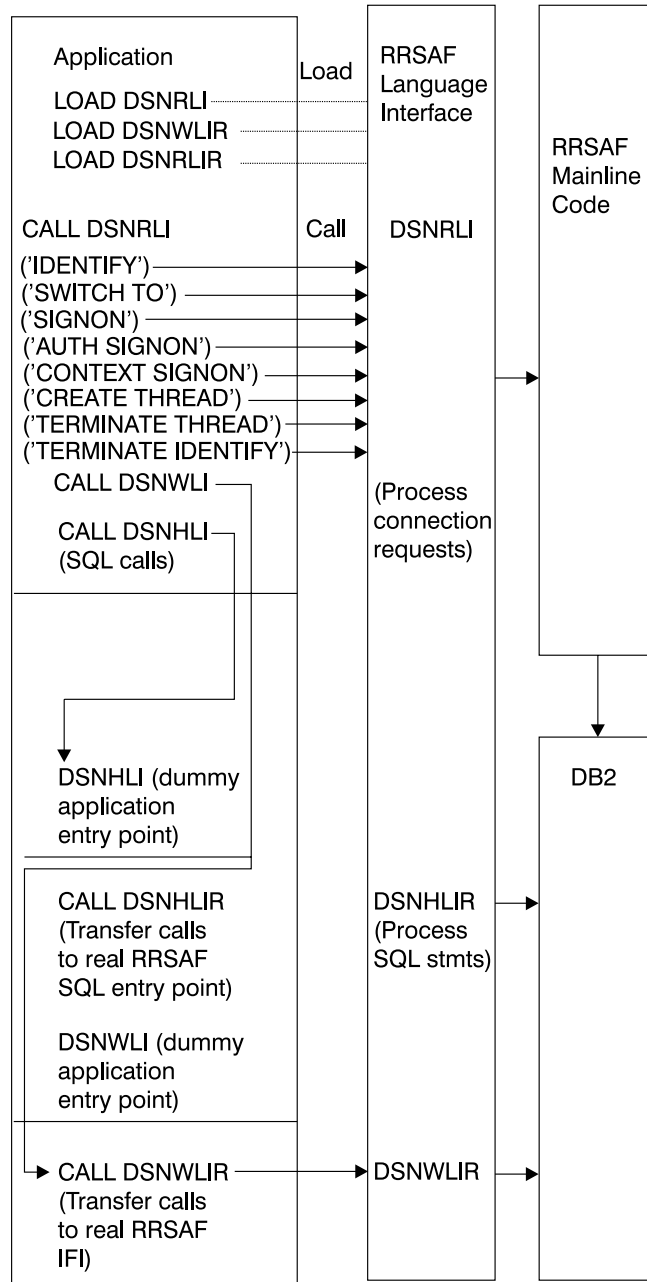
Figure 223. Sample RRSAF configuration

Part of RRSAF is a DB2 load module, DSNRLI, the RRSAF language interface module. DSNRLI has the alias names DSNHLIR and DSNWLIR. The module has five entry points: DSNRLI, DSNHLI, DSNHLIR, DSNWLI, and DSNWLIR:

- Entry point DSNRLI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLIR handle SQL calls. Use DSNHLI if your application program link-edits RRSAF; use DSNHLIR if your application program loads RRSAF.
- DSNWLI and DSNWLIR handle IFI calls. Use DSNWLI if your application program link-edits RRSAF; use DSNWLIR if your application program loads RRSAF.

You can access the DSNRLI module by explicitly issuing LOAD requests when your program runs, or by including the DSNRLI module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Loading DSNRLI explicitly

To load DSNRLI, issue MVS LOAD macros for entry points DSNRLI and DSNHLIR. If you use IFI services, you must also load DSNWLIR. Save the entry point address that LOAD returns and use it in the CALL macro.

By explicitly loading the DSNRLI module, you can isolate the maintenance of your application from future IBM service to the language interface. If the language interface changes, the change will probably not affect your load module.

You must indicate to DB2 which entry point to use. You can do this in one of two ways:

- Specify the precompiler option ATTACH(RRSAF).

  This causes DB2 to generate calls that specify entry point DSNHLIR. You cannot use this option if your application is written in FORTRAN.

- Code a dummy entry point named DSNHLI within your load module.

  If you do not specify the precompiler option ATTACH, the DB2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code corresponding to the dummy entry point must preserve the option list passed in R1 and call DSNHLIR specifying the same option list. For a coding example of a dummy DSNHLI entry point, see "Using dummy entry point DSNHLI" on page 796.

Link-editing DSNRLI

You can include DSNRLI when you link-edit your load module. For example, you can use a linkage editor control statement like this in your JCL:

```
INCLUDE DB2LIB(DSNRLI).
```

By coding this statement, you avoid linking the wrong language interface module.

When you include DSNRLI during the link-edit, you do not include a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNRLI contains an entry point for DSNHLI, which is identical to DSNHLIR, and an entry point DSNWLI, which is identical to DSNWLIR.

A disadvantage of link-editing DSNRLI into your load module is that if IBM makes a change to DSNRLI, you must link-edit your program again.

General properties of RRSAF connections

Some of the basic properties of an RRSAF connection with DB2 are:

Connection name and connection type: The connection name and connection type are RRSAF. You can use the DISPLAY THREAD command to list RRSAF applications that have the connection name RRSAF.

Authorization id: Each DB2 connection is associated with a set of authorization IDs. A connection must have a primary ID, and can have one or more secondary IDs. Those identifiers are used for:

- Validating access to DB2
- Checking privileges on DB2 objects

- Assigning ownership of DB2 objects
- Identifying the user of a connection for audit, performance, and accounting traces.

RRSAF relies on the MVS System Authorization Facility (SAF) and a security product, such as RACF, to verify and authorize the authorization IDs. An application that connects to DB2 through RRSAF must pass those identifiers to SAF for verification and authorization checking. RRSAF retrieves the identifiers from SAF.

A location can provide an authorization exit routine for a DB2 connection to change the authorization IDs and to indicate whether the connection is allowed. The actual values assigned to the primary and secondary authorization IDs can differ from the values provided by a SIGNON or AUTH SIGNON request. A site's DB2 signon exit routine can access the primary and secondary authorization IDs and can modify the IDs to satisfy the site's security requirements. The exit can also indicate whether the signon request should be accepted.

For information about authorization IDs and the connection and signon exit routines, see Appendix B (Volume 2) of *DB2 Administration Guide*.

Scope: The RRSAF processes connections as if each task is entirely isolated. When a task requests a function, RRSAF passes the function to DB2, regardless of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem have access to the connection status of multiple tasks in an address space.

Do not mix RRSAF connections with other connection types in a single address space. The first connection to DB2 made from an address space determines the type of connection allowed.

Task termination
If an application that is connected to DB2 through RRSAF terminates normally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, then OS/390 RRS commits any changes made after the last commit point.

If the application terminates abnormally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, then OS/390 RRS rolls back any changes made after the last commit point.

In either case, DB2 deallocates the plan, if necessary, and terminates the application's connection.

DB2 abend
If DB2 abends while an application is running, DB2 rolls back changes to the last commit point. If DB2 terminates while processing a commit request, DB2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

Summary of connection functions

You can use the following functions with CALL DSNRLI:

IDENTIFY
> Establishes the task as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is initialized as a user of DB2. See "IDENTIFY: Syntax and usage" on page 771.

SWITCH TO
> Directs RRSAF, SQL or IFI requests to a specified DB2 subsystem. See "SWITCH TO: Syntax and usage" on page 774.

SIGNON
> Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. See "SIGNON: Syntax and usage" on page 776.

AUTH SIGNON
> Provides to DB2 a user ID, an Accessor Environment Element (ACEE) and, optionally, one or more secondary authorization IDs that are associated with the connection. See "AUTH SIGNON: Syntax and usage" on page 779.

CONTEXT SIGNON
> Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. You can execute CONTEXT SIGNON from an unauthorized program. See "CONTEXT SIGNON: Syntax and usage" on page 782.

CREATE THREAD
> Allocates a DB2 plan or package. CREATE THREAD must complete before the application can execute SQL statements. See "CREATE THREAD: Syntax and usage" on page 786.

TERMINATE THREAD
> Deallocates the plan. See "TERMINATE THREAD: Syntax and usage" on page 788 .

TERMINATE IDENTIFY
> Removes the task as a user of DB2 and, if this is the last or only task in the address space that has a DB2 connection, terminates the address space connection to DB2. See "TERMINATE IDENTIFY: Syntax and usage" on page 789.

TRANSLATE
> Returns an SQL code and printable text, in the SQLCA, that describes a DB2 error reason code. You cannot call the TRANSLATE function from the FORTRAN language. See "Translate: Syntax and usage" on page 790.

RRSAF function descriptions

To code RRSAF functions in C, COBOL, FORTRAN, or PL/I, follow the individual language's rules for making calls to assembler language routines. Specify the return code and reason code parameters in the parameter list for each RRSAF call.

This section contains the following information:

Register conventions

Table 89 summarizes the register conventions for RRSAF calls.

If you do not specify the return code and reason code parameters in your RRSAF calls, RRSAF puts a return code in register 15 and a reason code in register 0. If you specify the return code and reason code parameters, RRSAF places the return code in register 15 and in the return code parameter to accommodate high-level languages that support special return code processing. RRSAF preserves the contents of registers 2 through 14.

Table 89. Register conventions for RRSAF calls

Register	Usage
R1	Parameter list pointer
R13	Address of caller's save area
R14	Caller's return address
R15	RRSAF entry point address

Parameter conventions for function calls

For assembler language: Use a standard parameter list for an MVS CALL. This means that when you issue the call, register 1 must contain the address of a list of pointers to the parameters. Each pointer is a 4-byte address. The last address must contain the value 1 in the high-order bit.

In an assembler language call, code a comma for a parameter in the CALL DSNRLI statement when you want to use the default value for that parameter and specify subsequent parameters. For example, code an IDENTIFY call like this to specify all optional parameters except *Return Code*:

```
CALL  DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,,REASCODE)
```

For all languages: When you code CALL DSNRLI statements in any language, specify all parameters that come before *Return Code*. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters.

All parameters starting with *Return Code* are optional.

For all languages except assembler language: Code 0 for an optional parameter in the CALL DSNRLI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are coding an IDENTIFY call in a COBOL program. You want to specify all parameters except *Return Code*. Write the call in this way:

```
CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERMCB STARTECB
   BY CONTENT ZERO BY REFERENCE REASCODE.
```

IDENTIFY: Syntax and usage

IDENTIFY initializes a connection to DB2.

```
►►──CALL DSNRLI──(──function, ssnm, ribptr, eibptr, termecb, startecb──────────────►

  ►─────────────────────────────────────────────)──────────────────────────────►◄
     └─,retcode─┐
               └─,reascode─┐
                          └─,groupoverride─┘
```
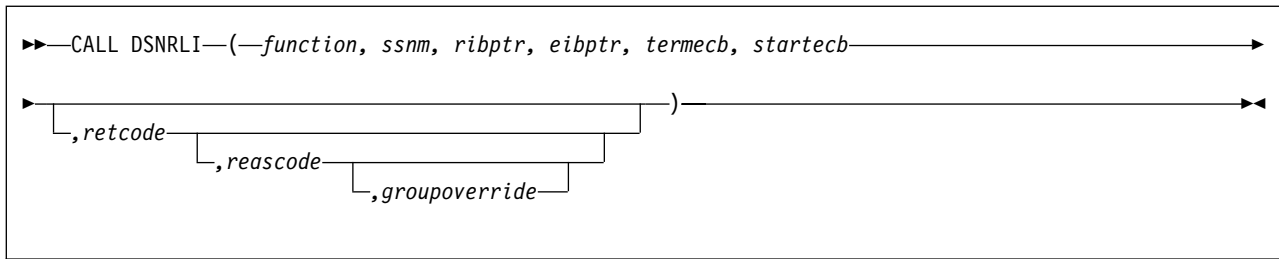
Figure 224. DSNRLI IDENTIFY function

Parameters point to the following areas:

function
   An 18-byte area containing IDENTIFY followed by 10 blanks.

ssnm
   A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made. If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

ribptr
   A 4-byte area in which RRSAF places the address of the release information block (RIB) after the call. This can be used to determine the release level of the DB2 subsystem to which the application is connected. You can determine the modification level within the release level by examining fields RIBCNUMB and RIBCINFO. If the value in RIBCNUMB is greater than zero, check RIBCINFO for modification levels.

   If the RIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

   The area to which *ribptr* points is below the 16-megabyte line.

   This parameter is required, although the application does not need to refer to the returned information.

eibptr
   A 4-byte area in which RRSAF places the address of the environment information block (EIB) after the call. The EIB contains environment information, such as the data sharing group and member name for the DB2 to which the IDENTIFY request was issued. If the DB2 subsystem is not in a data sharing group, then RRSAF sets the data sharing group and member names to blanks. If the EIB is not available (for example, if *ssnm* names a subsystem that does not exist), RRSAF sets the 4-byte area to zeros.

   The area to which *eibptr* points is above the 16-megabyte line.

   This parameter is required, although the application does not need to refer to the returned information.

termecb
   The address of the application's event control block (ECB) used for DB2 termination. DB2 posts this ECB when the system operator enters the command STOP DB2 or when DB2 is terminating abnormally. Specify a value of 0 if you do not want to use a termination ECB.

   RRSAF puts a POST code in the ECB to indicate the type of termination as shown in Table 90 on page 773.

Table 90. Post codes for types of DB2 termination

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

startecb
> The address of the application's startup ECB. If DB2 has not started when the application issues the IDENTIFY call, DB2 posts the ECB when DB2 startup has completed. Enter a value of zero if you do not want to use a startup ECB. DB2 posts a maximum of one startup ECB per address space. The ECB posted is associated with the most recent IDENTIFY call from that address space. The application program must examine any nonzero RRSAF or DB2 reason codes before issuing a WAIT on this ECB.

retcode
> A 4-byte area in which RRSAF places the return code.

> This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
> A 4-byte area in which RRSAF places a reason code.

> This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

> If you specify this parameter, you must also specify *retcode*.

groupoverride
> An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage: IDENTIFY establishes the caller's task as a user of DB2 services. If no other task in the address space currently is connected to the subsystem named by *ssnm*, then IDENTIFY also initializes the address space to communicate with the DB2 address spaces. IDENTIFY establishes the cross-memory authorization of the address space to DB2 and builds address space control blocks.

During IDENTIFY processing, DB2 determines whether the user address space is authorized to connect to DB2. DB2 invokes the MVS SAF and passes a primary authorization ID to SAF. That authorization ID is the 7-byte user ID associated with the address space, unless an authorized function has built an ACEE for the address space. If an authorized function has built an ACEE, DB2 passes the 8-byte user ID from the ACEE. SAF calls an external security product, such as RACF, to determine if the task is authorized to use:
- The DB2 resource class (CLASS=DSNR)
- The DB2 subsystem (SUBSYS=*ssnm*)
- Connection type RRSAF

If that check is successful, DB2 calls the DB2 connection exit to perform additional verification and possibly change the authorization ID. DB2 then sets the connection name to RRSAF and the connection type to RRSAF.

In a data sharing environment, use the *groupoverride* parameter on an IDENTIFY call when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

Table 91 shows an IDENTIFY call in each language.

Table 91. Examples of RRSAF IDENTIFY calls

Language	Call example
Assembler	CALL DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER)
C	fnret=dsnrli(&idfyfn[0],&ssnm[0], &ribptr, &eibptr, &termecb, &startecb, &retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNRLI' USING IDFYFN SSNM RIBTPR EIBPTR TERMECB STARTECB RETCODE REASCODE GRPOVER.
FORTRAN	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
             int DSNALI(
                char * functn,
                  ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

SWITCH TO: Syntax and usage

You can use SWITCH TO to direct RRSAF, SQL and/or IFI requests to a specified DB2 subsystem.

SWITCH TO is useful only after a successful IDENTIFY call. If you have established a connection with one DB2 subsystem, then you must issue SWITCH TO before you make an IDENTIFY call to another DB2 subsystem.

Figure 225. DSNRLI SWITCH TO function

Parameters point to the following areas:

function
    An 18-byte area containing SWITCH TO followed by nine blanks.

ssnm
    A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made. If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

retcode
    A 4-byte area in which RRSAF places the return code.

    This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
    A 4-byte area in which RRSAF places the reason code.

    This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

    If you specify this parameter, you must also specify *retcode*.

groupoverride
    An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage: Use SWITCH TO to establish connections to multiple DB2 subsystems from a single task. If you make a SWITCH TO call to a DB2 subsystem to which you have not issued an IDENTIFY call, DB2 returns return Code 4 and reason code X'00C12205' as a warning that the task has not yet identified to any DB2 subsystem.

After you establish a connection to a DB2 subsystem, you must make a SWITCH TO call before you identify to another DB2 subsystem. If you do not make a SWITCH TO call before you make an IDENTIFY call to another DB2 subsystem, then DB2 returns return Code = X'200' and reason code X'00C12201'.

In a data sharing environment, use the *groupoverride* parameter on an SWITCH TO call when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

This example shows how you can use SWITCH TO to interact with three DB2 subsystems.

```
RRSAF calls for subsystem db21:
  IDENTIFY
  SIGNON
  CREATE THREAD
Execute SQL on subsystem db21
SWITCH TO db22
RRSAF calls on subsystem db22:
  IDENTIFY
  SIGNON
```

```
             CREATE THREAD
         Execute SQL on subsystem db22
         SWITCH TO db23
         RRSAF calls on subsystem db23:
           IDENTIFY
           SIGNON
           CREATE THREAD
         Execute SQL on subsystem 23
         SWITCH TO db21
         Execute SQL on subsystem 21
         SWITCH TO db22
         Execute SQL on subsystem 22
         SWITCH TO db21
         Execute SQL on subsystem 21
         SRRCMIT (to commit the UR)
         SWITCH TO db23
         Execute SQL on subsystem 23
         SWITCH TO db22
         Execute SQL on subsystem 22
         SWITCH TO db21
         Execute SQL on subsystem 21
         SRRCMIT (to commit the UR)
```

Table 92 shows a SWITCH TO call in each language.

Table 92. Examples of RRSAF SWITCH TO calls

Language	Call example
Assembler	CALL DSNRLI,(SWITCHFN,SSNM,RETCODE,REASCODE,GRPOVER)
C	fnret=dsnrli(&switchfn[0], &ssnm[0], &retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNRLI' USING SWITCHFN RETCODE REASCODE GRPOVER.
FORTRAN	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C⁺⁺, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
                int DSNALI(
                  char * functn,
                    ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

SIGNON: Syntax and usage

SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for a connection.

```
►►──CALL DSNRLI──(──function, correlation-id,──accounting-token, accounting-interval──────────────►

   ►──────────────────────────────────────────────────────────────────────────── )─────►◄
      └─,─retcode─┐                                                          ┘
                  └─,─reascode─┐                                          ┘
                               └─,─user─┐                              ┘
                                        └─,─appl─┐                  ┘
                                                 └─,─ws─┐        ┘
                                                       └─,─xid─┘
```
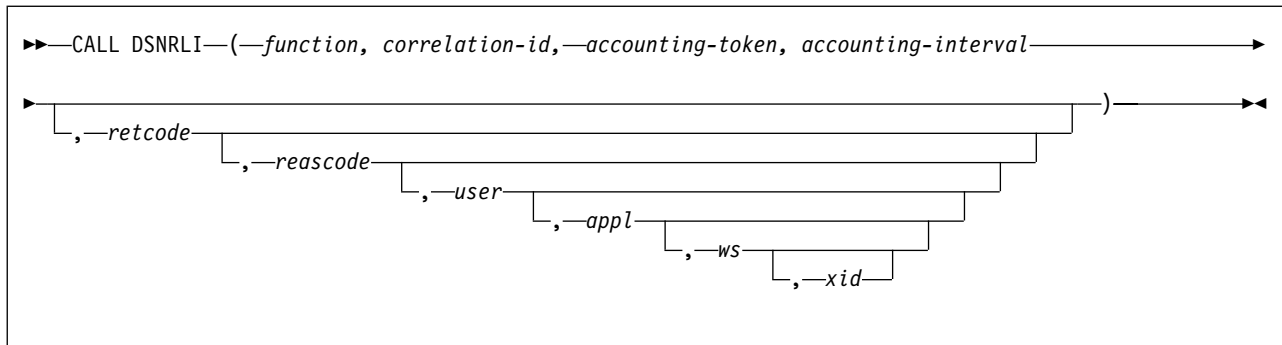
Figure 226. DSNRLI SIGNON function

Parameters point to the following areas:

function
> An 18-byte area containing SIGNON followed by twelve blanks.

correlation-id
> A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
> A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
> A 6-byte area with which you can control when DB2 writes an accounting record. If you specify COMMIT in that area, then DB2 writes an accounting record each time the application issues SRRCMIT. If you specify any other value, DB2 writes an accounting record when the application terminates or when you call SIGNON with a new authorization ID.

retcode
> A 4-byte area in which RRSAF places the return code.
>
> This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
> A 4-byte area in which RRSAF places the reason code.
>
> This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.
>
> If you specify this parameter, you must also specify *retcode*.

user
> A 16-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*. If not specified, no user ID is associated with the connection. You can omit this parameter by specifying a value of 0.

appl

A 32-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, and *user*. If not specified, no application or transaction is associated with the connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the workstation name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, *user*, and *appl*. If not specified, no workstation name is associated with the connection.

xid

A 4-byte area into which you put one of the following values:

0   Indicates that the thread is not part of a global transaction.

1   Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID.

address   The 4-byte address of of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify. The global transaction ID has the format shown in Table 93.

A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

Table 93. Format of a user-created global transaction ID

Field description	Length in bytes	Data type
Format ID	4	Character
Global transaction ID length	4	Integer
Branch qualifier length	4	Integer
Global transaction ID	1 to 64	Character
Branch qualifier	1 to 64	Character

Usage: SIGNON causes a new primary authorization ID and an optional secondary authorization IDs to be assigned to a connection. Your program does not need to be an authorized program to issue the SIGNON call. For that reason, before you issue the SIGNON call, you must issue the external security interface macro `RACROUTE REQUEST=VERIFY` to do the following:
- Define and populate an ACEE to identify the user of the program.
- Associate the ACEE with the user's TCB.
- Verify that the user is defined to RACF and authorized to use the application.

See *OS/390 Security Server (RACF) Macros and Interfaces* for more information on the RACROUTE macro.

Generally, you issue a SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a SIGNON call if the application is at a point of consistency, and
- The value of *reuse* in the CREATE THREAD call was RESET, or
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If there are open held cursors or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

Table 94 shows a SIGNON call in each language.

Table 94. Examples of RRSAF SIGNON calls

Language	Call example
assembler	CALL DSNRLI,(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&sgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING SGNONFN CORRID ACCTTKN ACCTINT RETCODE REASCODE USERID APPLNAME WSNAME.
FORTRAN	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
             int DSNALI(
               char * functn,
                 ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

AUTH SIGNON: Syntax and usage
AUTH SIGNON allows an APF-authorized program to pass either of the following to DB2:
- A primary authorization ID and, optionally, one or more secondary authorization IDs.
- An ACEE that is used for authorization checking

AUTH SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for the connection.

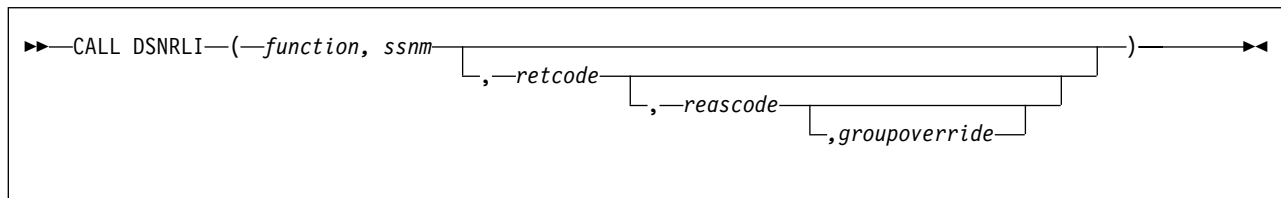*Figure 227. DSNRLI AUTH SIGNON function*

Parameters point to the following areas:

function
> An 18-byte area containing AUTH SIGNON followed by seven blanks.

correlation-id
> A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
> A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
> A 6-byte area with which you can control when DB2 writes an accounting record. If you specify COMMIT in that area, then DB2 writes an accounting record each time the application issues SRRCMIT. If you specify any other value, DB2 writes an accounting record when the application terminates or when you call SIGNON with a new authorization ID.

primary-authid
> An 8-byte area in which you can put a primary authorization ID. If you are not passing the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area.

ACEE-address
> The 4-byte address of an ACEE that you pass to DB2. If you do not want to provide an ACEE, specify 0 in this field.

secondary-authid
> An 8-byte area in which you can put a secondary authorization ID. If you do not pass the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area. If you enter a secondary authorization ID, you must also enter a primary authorization ID.

retcode
> A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

user

A 16-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*. If not specified, no user ID is associated with the connection. You can omit this parameter by specifying a value of 0.

appl

A 32-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, and *user*. If not specified, no application or transaction is associated with the connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the workstation name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, *user*, and *appl*. If not specified, no workstation name is associated with the connection.

xid

A 4-byte area into which you put one of the following values:

0   Indicates that the thread is not part of a global transaction.

1   Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID.

address   The 4-byte address of of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction

with the ID that you specify. The global transaction ID has the format shown in Table 93 on page 778.

A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

Usage: AUTH SIGNON causes a new primary authorization ID and optional secondary authorization IDs to be assigned to a connection.

Generally, you issue an AUTH SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue an AUTH SIGNON call if the application is at a point of consistency, and

- The value of *reuse* in the CREATE THREAD call was RESET, or
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If there are open held cursors or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

Table 95 shows a AUTH SIGNON call in each language.

Table 95. Examples of RRSAF AUTH SIGNON calls

Language	Call example
Assembler	CALL DSNRLI,(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&asgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &pauthid[0], &aceeptr, &sauthid[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING ASGNONFN CORRID ACCTTKN ACCTINT PAUTHID ACEEPTR SAUTHID RETCODE REASCODE USERID APPLNAME WSNAME.
FORTRAN	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
          int DSNALI(
            char * functn,
            ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

CONTEXT SIGNON: Syntax and usage
CONTEXT SIGNON establishes a primary authorization ID and one or more secondary authorization IDs for a connection.

```
►►──CALL DSNRLI──(──function, correlation-id, accounting-token,──accounting-interval, context-key────────►

►────────────────────────────────────────────────────────────────────────────)────►◄
     └─,─retcode─┐                                                 │
               └─,─reascode─┐                             │
                          └─,─user─┐                 │
                                 └─,─appl─┐       │
                                        └─,─ws─┐ │
                                             └─,─xid─┘
```
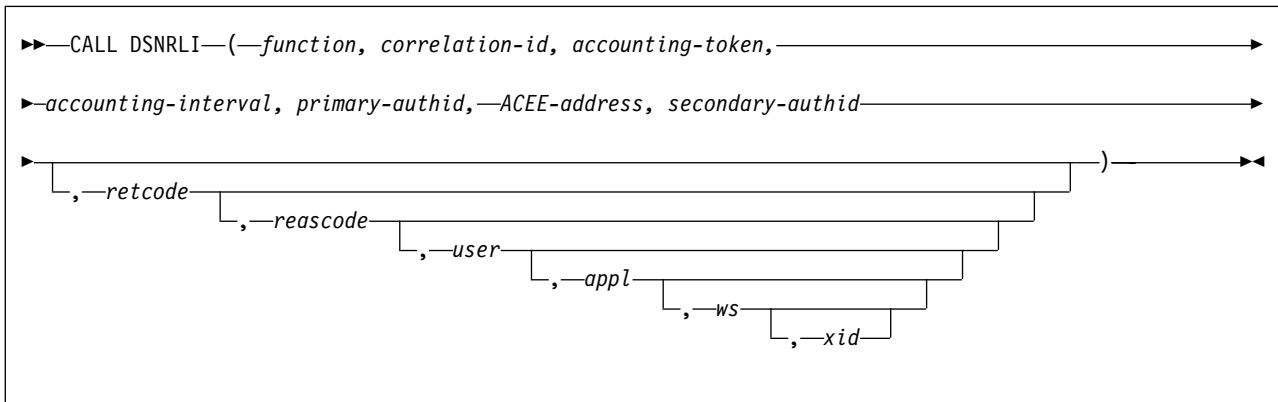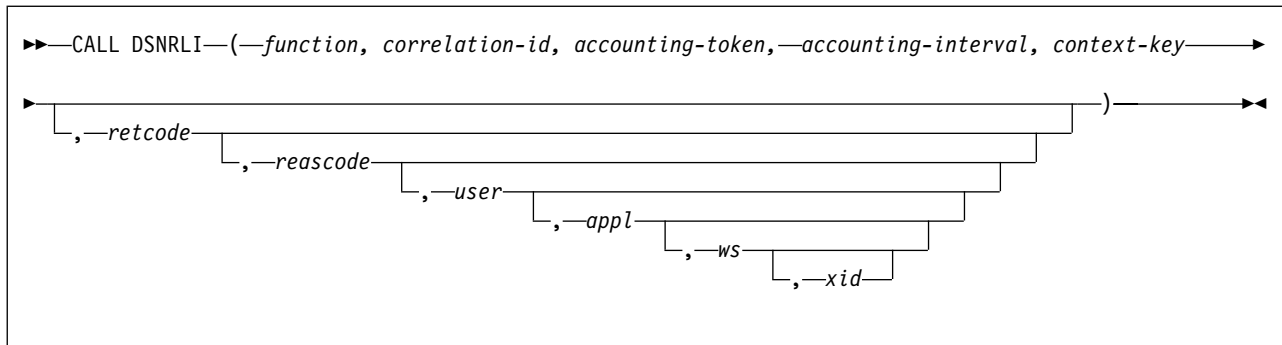
Figure 228. DSNRLI CONTEXT SIGNON function

Parameters point to the following areas:

function
An 18-byte area containing CONTEXT SIGNON followed by four blanks.

correlation-id
A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
A 6-byte area with which you can control when DB2 writes an accounting record. If you specify COMMIT in that area, then DB2 writes an accounting record each time the application issues SRRCMIT. If you specify any other value, DB2 writes an accounting record when the application terminates or when you call SIGNON with a new authorization ID.

context-key
A 32-byte area in which you put the context key that you specified when you called the RRS Set Context Data (CTXSDTA) service to save the primary authorization ID and an optional ACEE address.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

user
A 16-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output

and in DB2 accounting and statistics trace records. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*. If not specified, no user ID is associated with the connection. You can omit this parameter by specifying a value of 0.

appl
> A 32-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.
>
> This field is optional. If specified, you must also specify *retcode*, *reascode*, and *user*. If not specified, no application or transaction is associated with the connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the workstation name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

> This field is optional. If specified, you must also specify *retcode*, *reascode*, *user*, and *appl*. If not specified, no workstation name is associated with the connection.

xid
> A 4-byte area into which you put one of the following values:
>
> | | |
> |---|---|
> | **0** | Indicates that the thread is not part of a global transaction. |
> | **1** | Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID. |
> | *address* | The 4-byte address of of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify. The global transaction ID has the format shown in Table 93 on page 778. |
>
> A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

Usage: CONTEXT SIGNON relies on the RRS context services functions Set Context Data (CTXSDTA) and Retrieve Context Data (CTXRDTA). Before you invoke CONTEXT SIGNON, you must have called CTXSDTA to store a primary authorization ID and optionally, the address of an ACEE in the context data whose context key you supply as input to CONTEXT SIGNON.

CONTEXT SIGNON establishes a new primary authorization ID for the connection and optionally causes one or more secondary authorization IDs to be assigned. CONTEXT SIGNON uses the context key to retrieve the primary authorization ID from data associated with the current RRS context. DB2 uses the RRS context services function CTXRDTA to retrieve context data that contains the authorization ID and ACEE address. The context data must have the following format:

Version Number
> A 4-byte area that contains the version number of the context data. Set this area to 1.

Server Product Name
> An 8-byte area that contains the name of the server product that set the context data.

ALET   A 4-byte area that can contain an ALET value. DB2 does not reference this area.

ACEE Address
> A 4-byte area that contains an ACEE address or 0 if an ACEE is not provided. DB2 requires that the ACEE is in the home address space of the task.

primary-authid An 8-byte area that contains the primary authorization ID to be used. If the authorization ID is less than 8 bytes in length, pad it on the right with blank characters to a length of 8 bytes.

If the new primary authorization ID is not different than the current primary authorization ID (established at IDENTIFY time or at a previous SIGNON invocation) then DB2 invokes only the signon exit. If the value has changed, then DB2 establishes a new primary authorization ID and new SQL authorization ID and then invokes the signon exit.

If you pass an ACEE address, then CONTEXT SIGNON uses the value in ACEEGRPN as the secondary authorization ID if the length of the group name (ACEEGRPL) is not 0.

Generally, you issue a CONTEXT SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a CONTEXT SIGNON call if the application is at a point of consistency, and
- The value of *reuse* in the CREATE THREAD call was RESET, or
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If there are open held cursors or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

Table 96 shows a CONTEXT SIGNON call in each language.

Table 96. Examples of RRSAF CONTEXT SIGNON calls

Language	Call example
Assembler	CALL DSNRLI,(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&csgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &ctxtkey[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING CSGNONFN CORRID ACCTTKN ACCTINT CTXTKEY RETCODE REASCODE USERID APPLNAME WSNAME.

Table 96. Examples of RRSAF CONTEXT SIGNON calls (continued)

Language	Call example
FORTRAN	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE, USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C⁺⁺, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
          int DSNALI(
            char * functn,
             ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

CREATE THREAD: Syntax and usage

CREATE THREAD allocates DB2 resources for the application.
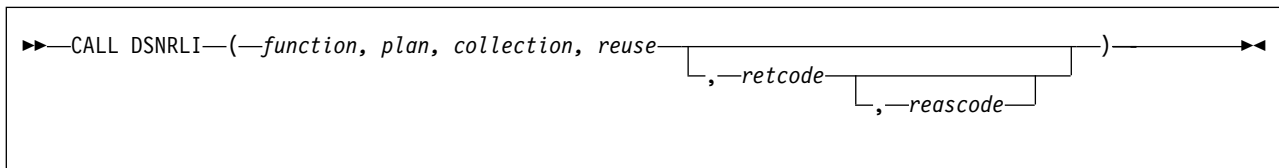
Figure 229. DSNRLI CREATE THREAD function

Parameters point to the following areas:

function
> An 18-byte area containing CREATE THREAD followed by five blanks.

plan
> An 8-byte DB2 plan name. If you provide a collection name instead of a plan name, specify the character ? in the first byte of this field. DB2 then allocates a special plan named ?RRSAF and uses the *collection* parameter. If you do not provide a collection name in the *collection* field, you must enter a valid plan name in this field.

collection
> An 18-byte area in which you enter a collection name. When you provide a collection name and put the character ? in the *plan* field, DB2 allocates a plan named ?RRSAF and a package list that contains two entries:
> • This collection name
> • An entry that contains * for the location, collection name, and package name
>
> If you provide a plan name in the *plan* field, DB2 ignores the value in this field.

reuse
> An 8-byte area that controls the action DB2 takes if a SIGNON call is issued after a CREATE THREAD call. Specify either of these values in this field:
> • RESET - to release any held cursors and reinitialize the special registers
> • INITIAL - to disallow the SIGNON
>
> This parameter is required. If the 8-byte area does not contain either RESET or INITIAL, then the default value is INITIAL.

retcode
> A 4-byte area in which RRSAF places the return code.
>
> This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
> A 4-byte area in which RRSAF places the reason code.
>
> This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.
>
> If you specify this parameter, you must also specify *retcode*.

Usage: CREATE THREAD allocates the DB2 resources required to issue SQL or IFI requests. If you specify a plan name, RRSAF allocates the named plan. If you specify ? in the first byte of the plan name and provide a collection name, DB2 allocates a special plan named ?RRSAF and a package list that contains the following entries:
- The collection name
- An entry that contains * for the location, collection ID, and package name

The collection name is used to locate a package associated with the first SQL statement in the program. The entry that contains *.*.* lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.

The application can use the SQL statement SET CURRENT PACKAGESET to change the collection ID that DB2 uses to locate a package.

When DB2 allocates a plan named ?RRSAF, DB2 checks authorization to execute the package in the same way as it checks authorization to execute a package from a requester other than DB2 for OS/390 and z/OS. See Part 3 (Volume 1) of *DB2 Administration Guide* for more information on authorization checking for package execution.

Table 97 shows a CREATE THREAD call in each language.

Table 97. Examples of RRSAF CREATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI,(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE)
C	fnret=dsnrli(&crthrdfn[0], &plan[0], &collid[0], &reuse[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING CRTHRDFN PLAN COLLID REUSE RETCODE REASCODE.
FORTRAN	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE)
PL/I	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C⁺⁺, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
                int DSNALI(
                  char * functn,
                    ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE THREAD: Syntax and usage

TERMINATE THREAD deallocates DB2 resources that were previously allocated for an application by CREATE THREAD.

```
►►──CALL DSNRLI──(──function,─────────────────────────────)──────────────────►◄
                        └─,──retcode─┘
                              └─,──reascode─┘
```
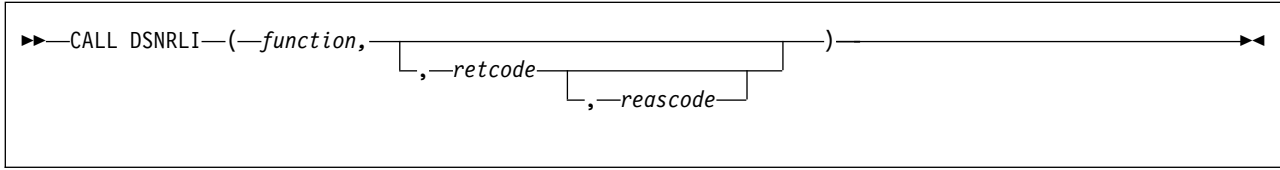
Figure 230. DSNRLI TERMINATE THREAD function

Parameters point to the following areas:

function
   An 18-byte area containing TERMINATE THREAD followed by two blanks.

retcode
   A 4-byte area in which RRSAF places the return code.

   This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
   A 4-byte area in which RRSAF places the reason code.

   This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

   If you specify this parameter, you must also specify *retcode*.

Usage: TERMINATE THREAD deallocates the DB2 resources associated with a plan. Those resources were previously allocated through CREATE THREAD. You can then use CREATE THREAD to allocate another plan using the same connection.

If you issue TERMINATE THREAD, and the application is not at a point of consistency, RRSAF returns reason code X'00C12211'.

Table 98 shows a TERMINATE THREAD call in each language.

Table 98. Examples of RRSAF TERMINATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI,(TRMTHDFN,RETCODE,REASCODE)
C	fnret=dsnrli(&trmthdfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TRMTHDFN RETCODE REASCODE.
FORTRAN	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE);

Table 98. Examples of RRSAF TERMINATE THREAD calls (continued)

Language	Call example

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
                   int DSNALI(
                       char * functn,
                        ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE IDENTIFY: Syntax and usage
TERMINATE IDENTIFY terminates a connection to DB2.
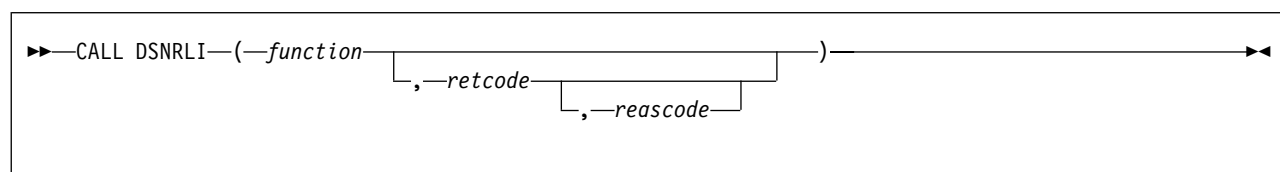
Figure 231. DSNRLI TERMINATE IDENTIFY function

Parameters point to the following areas:

function
An 18-byte area containing TERMINATE IDENTIFY.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: TERMINATE IDENTIFY removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross-memory authorization.

If the application is not at a point of consistency when you issue TERMINATE IDENTIFY, RRSAF returns reason code X'00C12211'.

If the application allocated a plan, and you issue TERMINATE IDENTIFY without first issuing TERMINATE THREAD, DB2 deallocates the plan before terminating the connection.

Issuing TERMINATE IDENTIFY is optional. If you do not, DB2 performs the same functions when the task terminates.

If DB2 terminates, the application must issue TERMINATE IDENTIFY to reset the RRSAF control blocks. This ensures that future connection requests from the task are successful when DB2 restarts.

Table 99 shows a TERMINATE IDENTIFY call in each language.

Table 99. Examples of RRSAF TERMINATE IDENTIFY calls

Language	Call example
Assembler	CALL DSNRLI,(TMIDFYFN,RETCODE,REASCODE)
C	fnret=dsnrli(&tmidfyfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TMIDFYFN RETCODE REASCODE.
FORTRAN	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C⁺⁺, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C⁺⁺   extern "OS" {
                int DSNALI(
                    char * functn,
                       ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

Translate: Syntax and usage

TRANSLATE converts a hexadecimal reason code for a DB2 error into a signed integer SQLCODE and a printable error message. The SQLCODE and message text are placed in the caller's SQLCA. You cannot call the TRANSLATE function from the FORTRAN language.

Issue TRANSLATE only after a successful IDENTIFY operation. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

```
►►──CALL DSNRLI──(──function, sqlca────────────────────)───────────►◄
                              └─,──retcode──┐
                                    └─,──reascode──┘
```
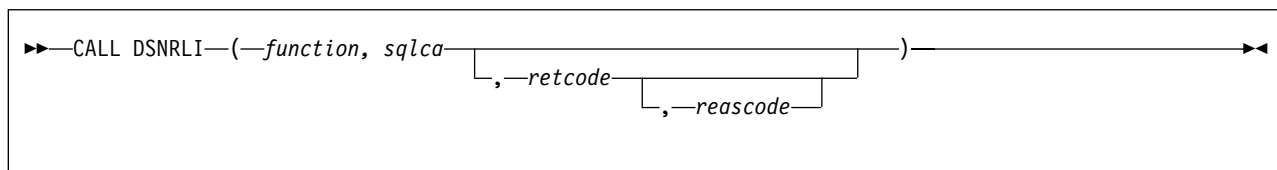
Figure 232. DSNRLI TRANSLATE function

Parameters point to the following areas:

function
    An 18-byte area containing the word TRANSLATE followed by nine blanks.

sqlca
    The program's SQL communication area (SQLCA).

retcode
    A 4-byte area in which RRSAF places the return code.

    This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
    A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that RRSAF returns in register 0 following a CREATE THREAD service request. DB2 places this information in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function translates codes that begin with X'00F3', but it does not translate RRSAF reason codes that begin with X'00C1'. If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original DB2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails; in which case, register 0 is set to X'00C12204' and register 15 is set to 200.

Table 100 shows a TRANSLATE call in each language.

Table 100. Examples of RRSAF TRANSLATE calls

Language	Call example
Assembler	CALL DSNRLI,(XLATFN,SQLCA,RETCODE,REASCODE)
C	fnret=dsnrli(&connfn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING XLATFN SQLCA RETCODE REASCODE.
PL/I	CALL DSNRLI(XLATFN,SQLCA,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C   #pragma linkage(dsnali, OS)

C++   extern "OS" {
         int DSNALI(
           char * functn,
            ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

Summary of RRSAF behavior

Table 101 on page 792 and Table 102 on page 792 summarize RRSAF behavior after various inputs from application programs. Errors are identified by the DB2 reason code that RRSAF returns. For a list of reason codes, see X'C1.' reason codes in Part 3 of *DB2 Messages and Codes*. Use these tables to understand the order in which your application must issue RRSAF calls, SQL statements, and IFI requests.

In these tables, the first column lists the most recent RRSAF or DB2 function executed. The first row lists the next function executed. The contents of the intersection of a row and column indicate the result of calling the function in the first column followed by the function in the first row. For example, if you issue TERMINATE THREAD, then you execute SQL or issue an IFI call, RRSAF returns reason code X'00C12219'.

Table 101. Effect of call order when next call is IDENTIFY, SWITCH TO, SIGNON, or CREATE THREAD

	Next function			
Previous function	IDENTIFY	SWITCH TO	SIGNON, AUTH SIGNON, or CONTEXT SIGNON	CREATE THREAD
Empty: first call	IDENTIFY	X'00C12205'	X'00C12204'	X'00C12204'
IDENTIFY	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12217'
SWITCH TO	IDENTIFY	Switch to *ssnm*	Signon [1]	CREATE THREAD
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	X'00C12201'	Switch to *ssnm*	Signon [1]	CREATE THREAD
CREATE THREAD	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'
TERMINATE THREAD	X'00C12201'	Switch to *ssnm*	Signon [1]	CREATE THREAD
IFI	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'
SQL	X'00C12201'	Switch to *ssnm*	X'00F30092' [2]	X'00C12202'
SRRCMIT or SRRBACK	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'

Notes:

1. Signon means the signon to DB2 through either SIGNON, AUTH SIGNON, or CONTEXT SIGNON.

2. SIGNON, AUTH SIGNON, or CONTEXT SIGNON are not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.

Table 102. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE

	Next function			
Previous function	SQL or IFI	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
Empty: first call	X'00C12204'	X'00C12204'	X'00C12204'	X'00C12204'
IDENTIFY	X'00C12218'	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
SWITCH TO	SQL or IFI call	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	X'00C12219'	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
CREATE THREAD	SQL or IFI call	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
TERMINATE THREAD	X'00C12219'	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
IFI	SQL or IFI call	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SQL	SQL or IFI call	X'00F30093' [1]	X'00F30093' [2]	TRANSLATE

Table 102. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE (continued)

SRRCMIT or SRRBACK	SQL or IFI call	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE

Notes:

1. TERMINATE THREAD is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.

2. TERMINATE IDENTIFY is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.

Sample scenarios

This section shows sample scenarios for connecting tasks to DB2.

A single task

This example shows a single task running in an address space. OS/390 RRS controls commit processing when the task terminates normally.

```
IDENTIFY
SIGNON
CREATE THREAD
SQL or IFI
  .
  .
  .
TERMINATE IDENTIFY
```

Multiple tasks

This example shows multiple tasks in an address space. Task 1 executes no SQL statements and makes no IFI calls. Its purpose is to monitor DB2 termination and startup ECBs and to check the DB2 release level.

```
TASK 1            TASK 2            TASK 3            TASK n

IDENTIFY          IDENTIFY          IDENTIFY          IDENTIFY
                  SIGNON            SIGNON            SIGNON
                  CREATE THREAD     CREATE THREAD     CREATE THREAD
                    SQL               SQL               SQL
                    ...               ...               ...
                  SRRCMIT           SRRCMIT           SRRCMIT
                  SQL               SQL               SQL
                    ...               ...               ...
                  SRRCMIT           SRRCMIT           SRRCMIT
                    ...               ...               ...
TERMINATE IDENTIFY
```

Calling SIGNON to reuse a DB2 thread

This example shows a DB2 thread that is to be used again by another user at a point of consistency. The application calls SIGNON for user B, using the DB2 plan that is allocated by the CREATE THREAD issued for user A.

```
IDENTIFY
SIGNON user A
CREATE THREAD
  SQL
  ...
  SRRCMIT
SIGNON user B
  SQL
  ...
  SRRCMIT
```

Switching DB2 threads between tasks

This example shows how you can switch the threads for four users (A, B, C, and D) among two tasks (1 and 2). The steps that the applications perform are:

- Task 1 creates context a, performs a context switch to make context a active for task 1, then identifies to a subsystem. A task must always perform an identify operation before a context switch can occur. After the identify operation is complete, task 1 allocates a thread for user A and performs SQL operations.

  At the same time, task 2 creates context b, performs a context switch to make context b active for task 2, identifies to the subsystem, then allocates a thread for user B and also performs SQL operations.

  When the SQL operations complete, both tasks perform OS/390 RRS context switch operations. Those operations disconnect each DB2 thread from the task under which it was running.

- Task 1 then creates context c, identifies to the subsystem, performs a context switch to make context c active for task 1, then allocates a thread for user C and performs SQL operations for user C.

  Task 2 does the same for user D.

  When the SQL operations for user C complete, task 1 performs a context switch operation to:
  - Switch the thread for user C away from task 1.
  - Switch the thread for user B to task 1.

  For a context switch operation to associate a task with a DB2 thread, the DB2 thread must have previously performed an identify operation. Therefore, before the thread for user B can be associated with task 1, task 1 must have performed an identify operation.

- Task 2 performs two context switch operations to:
  - Disassociate the thread for user D from task 2.
  - Associate the thread for user A with task 2.

```
Task 1                              Task 2

CTXBEGC (create context a)          CTXBEGC (create context b)
  CTXSWCH(a,0)                        CTXSWCH(b,0)
  IDENTIFY                            IDENTIFY
  SIGNON user A                       SIGNON user B
  CREATE THREAD (Plan A)              CREATE THREAD (plan B)
    SQL                                 SQL
    ...                                 ...
CTXSWCH(0,a)                         CTXSWCH(0,b)


CTXBEGC (create context c)          CTXBEGC (create context d)
  CTXSWCH(c,0)                        CTXSWCH(d,0)
  IDENTIFY                            IDENTIFY
  SIGNON user C                       SIGNON user D
  CREATE THREAD (plan C)              CREATE THREAD (plan D)
    SQL                                 SQL
    ...                                 ...
CTXSWCH(b,c)                         CTXSWCH(0,d)
    SQL (plan B)                        ...
    ...                             CTXSWCH(a,0)
                                        SQL (plan A)
```

RRSAF return codes and reason codes

If you specify return code and reason code parameters in your RRSAF call, RRSAF puts the return code and reason code in those parameters. Otherwise, RRSAF puts the return code in register 15 and the reason code in register 0. See Part 3 of *DB2 Messages and Codes* for detailed explanations of the reason codes.

When the reason code begins with X'00F3' (except for X'00F30006'), you can use the RRSAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, RRSAF returns standard SQL return codes in the SQLCA. See Part 1 of *DB2 Messages and Codes* for a list of those return codes and their meanings. RRSAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA). See Part 3 of *DB2 Messages and Codes* for a list of those return codes and their meanings.

Table 103. RRSAF return codes

Return code	Explanation
0	Successful completion.
4	Status information. See the reason code for details.
>4	The call failed. See the reason code for details.

Program examples

This section contains sample JCL for running an RRSAF application and assembler code for accessing RRSAF.

Sample JCL for using RRSAF

Use the sample JCL that follows as a model for using RRSAF in a batch environment. The DD statement for DSNRRSAF starts the RRSAF trace. Use that DD statement only if you are diagnosing a problem.

```
//jobname     JOB     MVS_jobcard_information
//RRSJCL      EXEC    PGM=RRS_application_program
//STEPLIB     DD      DSN=application_load_library
//            DD      DSN=DB2_load_library
    .
    .
    .
//SYSPRINT    DD      SYSOUT=*
//DSNRRSAF    DD      DUMMY
//SYSUDUMP    DD      SYSOUT=*
```

Loading and deleting the RRSAF language interface

The following code segment shows how an application loads entry points DSNRLI and DSNHLIR of the RRSAF language interface. Storing the entry points in variables LIRLI and LISQL ensures that the application loads the entry points only once.

Delete the loaded modules when the application no longer needs to access DB2.

```
***************************** GET LANGUAGE INTERFACE ENTRY ADDRESSES
        LOAD    EP=DSNRLI        Load the RRSAF service request EP
        ST      R0,LIRLI         Save this for RRSAF service requests
        LOAD    EP=DSNHLIR       Load the RRSAF SQL call Entry Point
        ST      R0,LISQL         Save this for SQL calls
```

```
*       .
*       .       Insert connection service requests and SQL calls here
*       .
        DELETE EP=DSNRLI          Correctly maintain use count
        DELETE EP=DSNHLIR         Correctly maintain use count
```

Using dummy entry point DSNHLI

Each of the DB2 attachment facilities contains an entry point named DSNHLI. When you use RRSAF but do not specify the precompiler option ATTACH(RRSAF), the precompiler generates BALR instructions to DSNHLI for SQL statements in your program. To find the correct DSNHLI entry point without including DSNRLI in your load module, code a subroutine, with entry point DSNHLI, that passes control to entry point DSNHLIR in the DSNRLI module. DSNHLIR is unique to DSNRLI and is at the same location as DSNHLI in DSNRLI. DSNRLI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, the intermediate subroutine must account for the difference.

In the example that follows, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
************************************************************************
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
************************************************************************
        DS    0D
DSNHLI  CSECT                     Begin CSECT
        STM   R14,R12,12(R13)     Prologue
        LA    R15,SAVEHLI         Get save area address
        ST    R13,4(,R15)         Chain the save areas
        ST    R15,8(,R13)         Chain the save areas
        LR    R13,R15             Put save area address in R13
        L     R15,LISQL           Get the address of real DSNHLI
        BASSM R14,R15             Branch to DSNRLI to do an SQL call
*                                 DSNRLI is in 31-bit mode, so use
*                                 BASSM to assure that the addressing
*                                 mode is preserved.
        L     R13,4(,R13)         Restore R13 (caller's save area addr)
        L     R14,12(,R13)        Restore R14 (return address)
        RETURN (1,12)             Restore R1-12, NOT R0 and R15 (codes)
```

Establishing a connection to DB2

Figure 233 on page 797 shows how to issue requests for certain RRSAF functions (IDENTIFY, SIGNON, CREATE THREAD, TERMINATE THREAD, and TERMINATE IDENTIFY).

The code in Figure 233 does not show a task that waits on the DB2 termination ECB. You can code such a task and use the MVS WAIT macro to monitor the ECB. The task that waits on the termination ECB should detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. The task in Figure 233 waits on the startup ECB at its own task level.

```
***************************** IDENTIFY *********************************
        L     R15,LIRLI           Get the Language Interface address
        CALL  (15),(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB),VL,MF=X
              (E,RRSAFCLL)
        BAL   R14,CHEKCODE        Call a routine (not shown) to check
*                                  return and reason codes
        CLC   CONTROL,CONTINUE    Is everything still OK
        BNE   EXIT                If CONTROL not 'CONTINUE', stop loop
        USING R8,RIB              Prepare to access the RIB
        L     R8,RIBPTR           Access RIB to get DB2 release level
        WRITE 'The current DB2 release level is' RIBREL
***************************** SIGNON ***********************************
        L     R15,LIRLI           Get the Language Interface address
        CALL  (15),(SGNONFN,CORRID,ACCTTKN,ACCTINT),VL,MF=(E,RRSAFCLL)
        BAL   R14,CHEKCODE        Check the return and reason codes
************************** CREATE THREAD *******************************
        L     R15,LIRLI           Get the Language Interface address
        CALL  (15),(CRTHRDFN,PLAN,COLLID,REUSE),VL,MF=(E,RRSAFCLL)
        BAL   R14,CHEKCODE        Check the return and reason codes
***************************** SQL **************************************
*             Insert your SQL calls here.  The DB2 Precompiler
*             generates calls to entry point DSNHLI.  You should
*             code a dummy entry point of that name to intercept
*             all SQL calls.  A dummy DSNHLI is shown below.
*********************** TERMINATE THREAD *******************************
        CLC   CONTROL,CONTINUE    Is everything still OK?
        BNE   EXIT                If CONTROL not 'CONTINUE', shut down
        L     R15,LIRLI           Get the Language Interface address
        CALL  (15),(TRMTHDFN),VL,MF=(E,RRSAFCLL)
        BAL   R14,CHEKCODE        Check the return and reason codes
*********************** TERMINATE IDENTIFY *****************************
        CLC   CONTROL,CONTINUE    Is everything still OK
        BNE   EXIT                If CONTROL not 'CONTINUE', stop loop
        L     R15,LIRLI           Get the Language Interface address
        CALL  (15),(TMIDFYFN),VL,MF=(E,RRSAFCLL)
        BAL   R14,CHEKCODE        Check the return and reason codes
```

Figure 233. Using RRSAF to connect to DB2

Figure 234 on page 798 shows declarations for some of the variables used in
Figure 233.

```
                 ****************** VARIABLES SET BY APPLICATION ***********************
                 LIRLI    DS    F               DSNRLI entry point address
                 LISQL    DS    F               DSNHLIR entry point address
                 SSNM     DS    CL4             DB2 subsystem name for IDENTIFY
                 CORRID   DS    CL12            Correlation ID for SIGNON
                 ACCTTKN  DS    CL22            Accounting token for SIGNON
                 ACCTINT  DS    CL6             Accounting interval for SIGNON
                 PLAN     DS    CL8             DB2 plan name for CREATE THREAD
                 COLLID   DS    CL18            Collection ID for CREATE THREAD.  If
                 *                              PLAN contains a plan name, not used.
                 REUSE    DS    CL8             Controls SIGNON after CREATE THREAD
                 CONTROL  DS    CL8             Action that application takes based
                 *                               on return code from RRSAF
                 ****************** VARIABLES SET BY DB2 ********************************
                 STARTECB DS    F               DB2 startup ECB
                 TERMECB  DS    F               DB2 termination ECB
                 EIBPTR   DS    F               Address of environment info block
                 RIBPTR   DS    F               Address of release info block
                 **************************** CONSTANTS *********************************
                 CONTINUE DC    CL8'CONTINUE'         CONTROL value: Everything OK
                 IDFYFN   DC    CL18'IDENTIFY         ' Name of RRSAF service
                 SGNONFN  DC    CL18'SIGNON          ' Name of RRSAF service
                 CRTHRDFN DC    CL18'CREATE THREAD    ' Name of RRSAF service
                 TRMTHDFN DC    CL18'TERMINATE THREAD ' Name of RRSAF service
                 TMIDFYFN DC    CL18'TERMINATE IDENTIFY' Name of RRSAF service
                 **************************** SQLCA and RIB *****************************
                     EXEC SQL INCLUDE SQLCA
                         DSNDRIB                 Map the DB2 Release Information Block
                 ****************** Parameter list for RRSAF calls *********************
                 RRSAFCLL CALL  ,(*,*,*,*,*,*,*,*),VL,MF=L
```

Figure 234. Declarations for variables used in the RRSAF connection routine

Chapter 31. Programming considerations for CICS

This section discusses some special topics of importance to CICS application programmers:
- Controlling the CICS attachment facility from an application
- Improving thread reuse
- Detecting whether the CICS attachment facility is operational

Controlling the CICS attachment facility from an application

You can start and stop the CICS attachment facility from within an application program. To start the attach facility, include this statement in your source code:

```
EXEC CICS LINK PROGRAM('DSN2COM0')
```

To stop the attachment facility, include this statement:

```
EXEC CICS LINK PROGRAM('DSN2COM2')
```

When you use this method, the attachment facility uses the default RCT. The default RCT name is DSN2CT concatenated with a one- or two-character suffix. The system administrator specifies this suffix in the DSN2STRT subparameter of the INITPARM parameter in the CICS startup procedure. If no suffix is specified, CICS uses an RCT name of DSN2CT00.

Improving thread reuse

In general, you want transactions to reuse threads whenever possible, because there is a high processor cost associated with thread creation. Part 5 (Volume 2) of *DB2 Administration Guide* contains a discussion of what factors affect CICS thread reuse and how you can write your applications to control these factors.

One of the most important things you can do to maximize thread reuse is to close all cursors that you declared WITH HOLD before each sync point, because DB2 does not automatically close them. A thread for an application that contains an open cursor cannot be reused. It is a good programming practice to close all cursors immediately after you finish using them. For more information on the effects of declaring cursors WITH HOLD in CICS applications, see "Held and non-held cursors" on page 91.

Detecting whether the CICS attachment facility is operational

You can use the INQUIRE EXITPROGRAM command in your applications to test whether the CICS attachment is available. The following example shows how to do this:

```
STST     DS     F
ENTNAME  DS     CL8
EXITPROG DS     CL8
 .
 .
 .
         MVC    ENTNAME,=CL8'DSNCSQL'
         MVC    EXITPROG,=CL8'DSN2EXT1'
         EXEC CICS INQUIRE EXITPROGRAM(EXITPROG)                      X
               ENTRYNAME(ENTNAME) STARTSTATUS(STST) NOHANDLE
         CLC    EIBRESP,DFHRESP(NORMAL)
         BNE    NOTREADY
         CLC    STST,DFHVALUE(CONNECTED)
         BNE    NOTREADY
```

```
UPNREADY DS    0H
        attach is up
NOTREADY DS    0H
        attach isn't up yet
```

In this example, the INQUIRE EXITPROGRAM command tests whether the
resource manager for SQL, DSNCSQL, is up and running. CICS returns the results
in the EIBRESP field of the EXEC interface block (EIB) and in the field whose name
is the argument of the STARTSTATUS parameter (in this case, STST). If the
EIBRESP value indicates that the command completed normally and the STST
value indicates that the resource manager is available, it is safe to execute SQL
statements. For more information on the INQUIRE EXITPROGRAM command, see
CICS for MVS/ESA System Programming Reference.

Attention

The *stormdrain* effect is a condition that occurs when a system continues to
receive work, even though that system is down.

When both of the following conditions are true, the stormdrain effect can
occur:
* The CICS attachment facility is down.
* You are using INQUIRE EXITPROGRAM to avoid AEY9 abends.

For more information on the stormdrain effect and how to avoid it, see Chapter
2 of *DB2 Data Sharing: Planning and Administration*.

If you are using a release of CICS after CICS Version 4, and you have specified
STANDBY=SQLCODE and STRTWT=AUTO in the DSNCRCT TYPE=INIT macro,
you do not need to test whether the CICS attachment facility is up before executing
SQL. When an SQL statement is executed, and the CICS attachment facility is not
available, DB2 issues SQLCODE -923 with a reason code that indicates that the
attachment facility is not available. See Part 2 of *DB2 Installation Guide* for
information about the DSNCRCT macro and *DB2 Messages and Codes* for an
explanation of SQLCODE -923.

Chapter 32. Programming techniques: Questions and answers

This chapter answers some frequently asked questions about database
programming techniques.

Providing a unique key for a table

Question: How can I provide a unique identifier for a table that has no unique
column?

Answer: Add a column with the data type ROWID or an identity column. ROWID
columns and identity columns contain a unique value for each row in the table. You
can define the column as GENERATED ALWAYS, which means that you cannot
insert values into the column, or GENERATED BY DEFAULT, which means that
DB2 generates a value if you do not specify one. If you define the ROWID or
identity column as GENERATED BY DEFAULT, you need to define a unique index
that includes only that column to guarantee uniqueness.

Scrolling through previously retrieved data

Question: When a program retrieves data from the database, how can the program
scroll backward through the data?

Answer: Use one of the following techniques:
- Use a scrollable cursor.
- If the table contains a ROWID or an identity column, retrieve the values from that
  column into an array. Then use the ROWID or identity column values to retrieve
  the rows in reverse order.

These options are described in more detail below.

Using a scrollable cursor

Using a scrollable cursor to fetch backward through data involves these basic steps:
1. Declare the cursor with the SCROLL parameter.
2. Open the cursor.
3. Execute a FETCH statement to position the cursor at the end of the result table.
4. In a loop, execute FETCH statements that move the cursor backward and then
   retrieve the data.
5. When you have retrieved all the data, close the cursor.

For example, you can use code like the following to retrieve department names in
reverse order from table DSN8710.DEPT:

```
/************************/
/* Declare host variables */
/************************/
EXEC SQL BEGIN DECLARE SECTION;
  char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*********************************************************/
/* Declare scrollable cursor to retrieve department names */
/*********************************************************/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
  SELECT DEPTNAME FROM DSN8710.DEPT;
  :
/*********************************************************/
```

```
                /* Open the cursor and position it after the end of the   */
                /* result table.                                           */
                /***********************************************************/
                EXEC SQL OPEN C1;
                EXEC SQL FETCH AFTER FROM C1;
                /***********************************************************/
                /* Fetch rows backward until all rows are fetched.         */
                /***********************************************************/
                while(SQLCODE==0) {
                  EXEC SQL FETCH PRIOR FROM C1 INTO :hv_deptname;

                  :
                  :
                }
                EXEC SQL CLOSE C1;
```

Using a ROWID or identity column

If your table contains a ROWID column or an identity column, you can use that
column to rapidly retrieve the rows in reverse order. When you perform the original
SELECT, you can store the ROWID or identity column value for each row you
retrieve. Then, to retrieve the values in reverse order, you can execute SELECT
statements with a WHERE clause that compares the ROWID or identity column
value to each stored value.

For example, suppose you add ROWID column DEPTROWID to table
DSN8710.DEPT. You can use code like the following to select all department
names, then retrieve the names in reverse order:

```
/**************************/
/* Declare host variables */
/**************************/
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS ROWID  hv_dept_rowid;
  char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/**************************/
/* Declare other variables */
/**************************/
struct rowid_struct {
  short int length;
  char data[40];    /* ROWID variable structure */
}
struct rowid_struct rowid_array[200];
                                /* Array to hold retrieved  */
                                /* ROWIDs.  Assume no more   */
                                /* than 200 rows will be     */
                                /* retrieved.                */
short int i,j,n;
/***********************************************/
/* Declare cursor to retrieve department names */
/***********************************************/
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNAME, DEPTROWID FROM DSN8710.DEPT;
  :
/***********************************************************/
/* Retrieve the department name and ROWID from DEPT table */
/* and store the ROWID in an array.                       */
/***********************************************************/
EXEC SQL OPEN C1;
i=0;
while(SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :hv_deptname, :hv_dept_rowid;
  rowid_array[i].length=hv_dept_rowid.length;
  for(j=0;j<hv_dept_rowid.length;j++)
    rowid_array[i].data[j]=hv_dept_rowid.data[j];
```

```
                          i++;
                        }
                      EXEC SQL CLOSE C1;
                      n=i-1;                  /* Get the number of array elements */
                      /***********************************************************/
                      /* Use the ROWID values to retrieve the department names   */
                      /* in reverse order.                                       */
                      /***********************************************************/
                      for(i=n;i>=0;i--) {
                        hv_dept_rowid.length=rowid_array[i].length;
                        for(j=0;j<hv_dept_rowid.length;j++)
                          hv_dept_rowid.data[j]=rowid_array[i].data[j];
                        EXEC SQL SELECT DEPTNAME INTO :hv_deptname
                          FROM DSN8710.DEPT
                            WHERE DEPTROWID=:hv_dept_rowid;
                      }
```

Scrolling through a table in any direction

Question: How can I fetch rows from a table in any direction?

Answer: Declare your cursor as scrollable. When you select rows from the table, you can use the various forms of the FETCH statement to move to an absolute row number, move ahead or back a certain number of rows, to the first or last row, before the first row or after the last row, forward, or backward. You can use any combination of these FETCH statements to change direction repeatedly.

For example, you can use code like the following to move forward in the department table by 10 records, backward five records, and forward again by three records:

```
/*************************/
/* Declare host variables */
/*************************/
EXEC SQL BEGIN DECLARE SECTION;
  char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/***********************************************************/
/* Declare scrollable cursor to retrieve department names */
/***********************************************************/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
  SELECT DEPTNAME FROM DSN8710.DEPT;
  .
  .
/***********************************************************/
/* Open the cursor and position it before the start of     */
/* the result table.                                       */
/***********************************************************/
EXEC SQL OPEN C1;
EXEC SQL FETCH BEFORE FROM C1;
/***********************************************************/
/* Fetch first 10 rows                                     */
/***********************************************************/
for(i=0;i<10;i++)
{
  EXEC SQL FETCH NEXT FROM C1 INTO :hv_deptname;
}
/***********************************************************/
/* Save the value in the tenth row                         */
/***********************************************************/
tenth_row=hv_deptname;
/***********************************************************/
/* Fetch backward 5 rows                                   */
/***********************************************************/
for(i=0;i<5;i++)
{
```

```
      EXEC SQL FETCH PRIOR FROM C1 INTO :hv_deptname;
    }
    /**********************************************************/
    /* Save the value in the fifth row                        */
    /**********************************************************/
    fifth_row=hv_deptname;
    /**********************************************************/
    /* Fetch forward 3 rows                                   */
    /**********************************************************/
    for(i=0;i<3;i++)
    {
      EXEC SQL FETCH NEXT FROM C1 INTO :hv_deptname;
    }
    /**********************************************************/
    /* Save the value in the eighth row                       */
    /**********************************************************/
    eighth_row=hv_deptname;
    /**********************************************************/
    /* Close the cursor                                       */
    /**********************************************************/

    EXEC SQL CLOSE C1;
```

Updating data as it is retrieved from the database

Question: How can I update rows of data as I retrieve them?

Answer: On the SELECT statement, use the FOR UPDATE clause without a column list, or the FOR UPDATE OF clause with a column list. For a more efficient program, specify a column list with only those columns that you intend to update. Then use the positioned UPDATE statement. The clause WHERE CURRENT OF identifies the cursor that points to the row you want to update.

Updating previously retrieved data

Question: How can you scroll backward and update data that was retrieved previously?

Answer: Use a scrollable cursor that is declared with the FOR UPDATE OF clause. Using a scrollable cursor to update backward involves these basic steps:

1. Declare the cursor with the SENSITIVE STATIC SCROLL parameters.
2. Open the cursor.
3. Execute a FETCH statement to position the cursor at the end of the result table.
4. FETCH statements that move the cursor backward, until you reach the row that you want to update.
5. Execute the UPDATE WHERE CURRENT OF statement to update the current row.
6. Repeat steps 4 and 5 until you have update all the rows that you need to.
7. When you have retrieved and updated all the data, close the cursor.

Updating thousands of rows

Question: Are there any special techniques for updating large volumes of data?

Answer: Yes. When updating large volumes of data using a cursor, you can minimize the amount of time that you hold locks on the data by declaring the cursor with the HOLD option and by issuing commits frequently.

Retrieving thousands of rows

Question: Are there any special techniques for fetching and displaying large volumes of data?

Answer: There are no special techniques; but for large numbers of rows, efficiency can become very important. In particular, you need to be aware of locking considerations, including the possibilities of lock escalation.

If your program allows input from a terminal before it commits the data and thereby releases locks, it is possible that a significant loss of concurrency results. Review the description of locks in "The ISOLATION option" on page 341 while designing your program. Then review the expected use of tables to predict whether you could have locking problems.

Using SELECT *

Question: What are the implications of using SELECT * ?

Answer: Generally, you should select only the columns you need because DB2 is sensitive to the number of columns selected. Use SELECT * only when you are sure you want to select all columns. One alternative is to use views defined with only the necessary columns, and use SELECT * to access the views. Avoid SELECT * if all the selected columns participate in a sort operation (SELECT DISTINCT and SELECT...UNION, for example).

Optimizing retrieval for a small set of rows

Question: How can I tell DB2 that I want only a few of the thousands of rows that satisfy a query?

Answer: Use OPTIMIZE FOR *n* ROWS or FETCH FIRST *n* ROWS ONLY.

DB2 usually optimizes queries to retrieve all rows that qualify. But sometimes you want to retrieve only the first few rows. For example, to retrieve the first row that is greater than or equal to a known value, code:

```
SELECT column list FROM table
  WHERE key >= value
  ORDER BY key ASC
```

Even with the ORDER BY clause, DB2 might fetch all the data first and sort it afterwards, which could be wasteful. Instead, you can write the query in one of the following ways:

```
SELECT * FROM table
  WHERE key >= value
  ORDER BY key ASC
  OPTIMIZE FOR 1 ROW
```
```
SELECT * FROM table
  WHERE key >= value
  ORDER BY key ASC
  FETCH FIRST n ROWS ONLY
```

Use OPTIMIZE FOR 1 ROW to influence the access path. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly.

Use FETCH FIRST *n* ROWS ONLY to limit the number of rows in the result table to *n* rows. FETCH FIRST *n* ROWS ONLY has the following benefits:

• When you use FETCH statements to retrieve data from a result table, FETCH FIRST n ROWS ONLY causes DB2 to retrieve only the number of rows that you need. This can have performance benefits, especially in distributed applications. If you try to execute a FETCH statement to retrieve the *n+1*st row, DB2 returns a +100 SQLCODE.

• When you use FETCH FIRST ROW ONLY in a SELECT INTO statement, you never retrieve more than one row. Using FETCH FIRST ROW ONLY in a SELECT INTO statement can prevent SQL errors that are caused by inadvertently selecting more than one value into a host variable.

When you specify FETCH FIRST *n* ROWS ONLY but not OPTIMIZE FOR *n* ROWS, OPTIMIZE FOR *n* ROWS is implied. When you specify FETCH FIRST *n* ROWS ONLY and OPTIMIZE FOR *m* ROWS, and *m* is less than *n*, DB2 optimizes the query for *m* rows. If *m* is greater than *n*, DB2 optimizes the query for *n* rows.

Adding data to the end of a table

Question: How can I add data to the end of a table?

Answer: Though the question is often asked, it has no meaning in a relational database. The rows of a base table are not ordered; hence, the table does not have an "end".

To get the effect of adding data to the "end" of a table, define a unique index on a TIMESTAMP column in the table definition. Then, when you retrieve data from the table, use an ORDER BY clause naming that column. The newest insert appears last.

Translating requests from end users into SQL statements

Question: A program translates requests from end users into SQL statements before executing them, and users can save a request. How can the corresponding SQL statement be saved?

Answer: You can save the corresponding SQL statements in a table with a column having a data type of VARCHAR(*n*), where *n* is the maximum length of any SQL statement. You must save the source SQL statements, not the prepared versions. That means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your program prepares an SQL statement from a character string and executes it dynamically. (For a description of dynamic SQL, see "Chapter 23. Coding dynamic SQL in application programs" on page 495.)

Changing the table definition

Question: How can I write an SQL application that allows users to create new tables, add columns to them, increase the length of character columns, rearrange the columns, and delete columns?

Answer: Your program can dynamically execute CREATE TABLE and ALTER TABLE statements entered by users to create new tables, add columns to existing tables, or increase the length of VARCHAR columns. Added columns initially contain either the null value or a default value. Both statements, like any data definition statement, are relatively expensive to execute; consider the effects of locks.

It is not possible to rearrange or delete columns in a table without dropping the entire table. You can, however, create a view on the table, which includes only the columns you want, in the order you want. This has the same effect as redefining the table.

For a description of dynamic SQL execution, see "Chapter 23. Coding dynamic SQL in application programs" on page 495.

Storing data that does not have a tabular format

Question: How can I store a large volume of data that is not defined as a set of columns in a table?

Answer: You can store the data in a table in a VARCHAR column or a LOB column.

Finding a violated referential or check constraint

Question: When a referential or check constraint has been violated, how do I determine which one it is?

Answer: When you receive an SQL error because of a constraint violation, print out the SQLCA. You can use the DSNTIAR routine described in "Handling SQL error return codes" on page 76 to format the SQLCA for you. Check the SQL error message insertion text (SQLERRM) for the name of the constraint. For information on possible violations, see SQLCODEs -530 through -548 in Part 1 of *DB2 Messages and Codes*.

Part 7. Appendixes

Appendix A. DB2 sample tables

Most of the examples in this book refer to the tables described in this appendix. As a group, the tables include information that describes employees, departments, projects, and activities, and make up a sample application that exemplifies most of the features of DB2. The sample storage group, databases, tablespaces, tables, and views are created when you run the installation sample jobs DSNTEJ1 and DSNTEJ7. DB2 sample objects that include LOBs are created in job DSNTEJ7. All other sample objects are created in job DSNTEJ1. The CREATE INDEX statements for the sample tables are not shown here; they, too, are created by the DSNTEJ1 and DSNTEJ7 sample jobs.

Authorization on all sample objects is given to PUBLIC in order to make the sample programs easier to run. The contents of any table can easily be reviewed by executing an SQL statement, for example SELECT * FROM DSN8710.PROJ. For convenience in interpreting the examples, the department and employee tables are listed here in full.

Activity table (DSN8710.ACT)

The activity table describes the activities that can be performed during a project. The table resides in database DSN8D71A and is created with:

```
CREATE TABLE DSN8710.ACT
      (ACTNO    SMALLINT      NOT NULL,
       ACTKWD   CHAR(6)       NOT NULL,
       ACTDESC  VARCHAR(20)   NOT NULL,
       PRIMARY KEY (ACTNO)              )
  IN DSN8D71A.DSN8S71P
  CCSID EBCDIC;
```

Content

Table 104 shows the content of the columns.

Table 104. Columns of the activity table

Column	Column Name	Description
1	ACTNO	Activity ID (the primary key)
2	ACTKWD	Activity keyword (up to six characters)
3	ACTDESC	Activity description

The activity table has these indexes:

Table 105. Indexes of the activity table

Name	On Column	Type of Index
DSN8710.XACT1	ACTNO	Primary, ascending
DSN8710.XACT2	ACTKWD	Unique, ascending

Relationship to other tables

The activity table is a parent table of the project activity table, through a foreign key on column ACTNO.

Department table (DSN8710.DEPT)

The department table describes each department in the enterprise and identifies its manager and the department to which it reports.

The table, shown in Table 108 on page 813, resides in table space DSN8D71A.DSN8S71D and is created with:

```
CREATE TABLE DSN8710.DEPT
      (DEPTNO    CHAR(3)          NOT NULL,
       DEPTNAME  VARCHAR(36)      NOT NULL,
       MGRNO     CHAR(6)                   ,
       ADMRDEPT  CHAR(3)          NOT NULL,
       LOCATION  CHAR(16)                  ,
       PRIMARY KEY (DEPTNO)              )
  IN DSN8D71A.DSN8S71D
  CCSID EBCDIC;
```

Because the table is self-referencing, and also is part of a cycle of dependencies, its foreign keys must be added later with these statements:

```
ALTER TABLE DSN8710.DEPT
      FOREIGN KEY RDD (ADMRDEPT) REFERENCES DSN8710.DEPT
              ON DELETE CASCADE;

ALTER TABLE DSN8710.DEPT
      FOREIGN KEY RDE (MGRNO) REFERENCES DSN8710.EMP
              ON DELETE SET NULL;
```

Content

Table 106 shows the content of the columns.

Table 106. Columns of the department table

Column	Column Name	Description
1	DEPTNO	Department ID, the primary key
2	DEPTNAME	A name describing the general activities of the department
3	MGRNO	Employee number (EMPNO) of the department manager
4	ADMRDEPT	ID of the department to which this department reports; the department at the highest level reports to itself
5	LOCATION	The remote location name

The department table has these indexes:

Table 107. Indexes of the department table

Name	On Column	Type of Index
DSN8710.XDEPT1	DEPTNO	Primary, ascending
DSN8710.XDEPT2	MGRNO	Ascending
DSN8710.XDEPT3	ADMRDEPT	Ascending

Relationship to other tables

The table is self-referencing: the value of the administering department must be a department ID.

The table is a parent table of:
- The employee table, through a foreign key on column WORKDEPT
- The project table, through a foreign key on column DEPTNO.

It is a dependent of the employee table, through its foreign key on column MGRNO.

Table 108. DSN8710.DEPT: department table

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	---------------
B01	PLANNING	000020	A00	---------------
C01	INFORMATION CENTER	000030	A00	---------------
D01	DEVELOPMENT CENTER	------	A00	---------------
E01	SUPPORT SERVICES	000050	A00	---------------
D11	MANUFACTURING SYSTEMS	000060	D01	---------------
D21	ADMINISTRATION SYSTEMS	000070	D01	---------------
E11	OPERATIONS	000090	E01	---------------
E21	SOFTWARE SUPPORT	000100	E01	---------------
F22	BRANCH OFFICE F2	------	E01	---------------
G22	BRANCH OFFICE G2	------	E01	---------------
H22	BRANCH OFFICE H2	------	E01	---------------
I22	BRANCH OFFICE I2	------	E01	---------------
J22	BRANCH OFFICE J2	------	E01	---------------

The LOCATION column contains nulls until sample job DSNTEJ6 updates this column with the location name.

Employee table (DSN8710.EMP)

The employee table identifies all employees by an employee number and lists basic personnel information.

The table shown in Table 111 on page 815 and Table 112 on page 816 resides in the partitioned table space DSN8D71A.DSN8S71E. Because it has a foreign key referencing DEPT, that table and the index on its primary key must be created first. Then EMP is created with:

```
CREATE TABLE DSN8710.EMP
     (EMPNO     CHAR(6)                        NOT NULL,
      FIRSTNME  VARCHAR(12)                     NOT NULL,
      MIDINIT   CHAR(1)                         NOT NULL,
      LASTNAME  VARCHAR(15)                     NOT NULL,
      WORKDEPT  CHAR(3)                                  ,
      PHONENO   CHAR(4)           CONSTRAINT NUMBER CHECK
                (PHONENO >= '0000' AND
                 PHONENO <= '9999')                      ,
      HIREDATE  DATE                                     ,
      JOB       CHAR(8)                                  ,
      EDLEVEL   SMALLINT                                 ,
      SEX       CHAR(1)                                  ,
      BIRTHDATE DATE                                     ,
      SALARY    DECIMAL(9,2)                             ,
      BONUS     DECIMAL(9,2)                             ,
      COMM      DECIMAL(9,2)                             ,
      PRIMARY KEY (EMPNO)                                ,
      FOREIGN KEY RED (WORKDEPT) REFERENCES DSN8710.DEPT
                ON DELETE SET NULL                       )
  EDITPROC  DSN8EAE1
  IN DSN8D71A.DSN8S71E
  CCSID EBCDIC;
```

Content

Table 109 shows the content of the columns. The table has a check constraint, NUMBER, which checks that the phone number is in the numeric range 0000 to 9999.

Table 109. Columns of the employee table

Column	Column Name	Description
1	EMPNO	Employee number (the primary key)
2	FIRSTNME	First name of employee
3	MIDINIT	Middle initial of employee
4	LASTNAME	Last name of employee
5	WORKDEPT	ID of department in which the employee works
6	PHONENO	Employee telephone number
7	HIREDATE	Date of hire
8	JOB	Job held by the employee
9	EDLEVEL	Number of years of formal education
10	SEX	Sex of the employee (M or F)
11	BIRTHDATE	Date of birth
12	SALARY	Yearly salary in dollars
13	BONUS	Yearly bonus in dollars
14	COMM	Yearly commission in dollars

The table has these indexes:

Table 110. Indexes of the employee table

Name	On Column	Type of Index
DSN8710.XEMP1	EMPNO	Primary, partitioned, ascending
DSN8710.XEMP2	WORKDEPT	Ascending

Relationship to other tables

The table is a parent table of:
- The department table, through a foreign key on column MGRNO
- The project table, through a foreign key on column RESPEMP.

It is a dependent of the department table, through its foreign key on column WORKDEPT.

*Table 111. Left half of DSN8710.EMP: employee table. Note that a blank in the MIDINIT column is an actual value of '
' rather than null.*

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10
000030	SALLY	A	KWAN	C01	4738	1975-04-05
000050	JOHN	B	GEYER	E01	6789	1949-08-17
000060	IRVING	F	STERN	D11	6423	1973-09-14
000070	EVA	D	PULASKI	D21	7831	1980-09-30
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16
000120	SEAN		O'CONNELL	A00	2167	1963-12-05
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15
000150	BRUCE		ADAMSON	D11	4510	1972-02-12
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07
000190	JAMES	H	WALKER	D11	2986	1974-07-26
000200	DAVID		BROWN	D11	4501	1966-03-03
000210	WILLIAM	T	JONES	D11	0942	1979-04-11
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1969-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5698	1947-05-05
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01
200120	GREG		ORLANDO	A00	2167	1972-05-05
200140	KIM	N	NATZ	C01	1793	1976-12-15
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15
200220	REBA	K	JOHN	D11	0672	1968-08-29
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12
200330	HELENA		WONG	E21	2103	1976-02-23
200340	ROY	R	ALONZO	E21	5698	1947-05-05

Table 112. Right half of DSN8710.EMP: employee table

(EMPNO)	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
(000010)	PRES	18	F	1933-08-14	52750.00	1000.00	4220.00
(000020)	MANAGER	18	M	1948-02-02	41250.00	800.00	3300.00
(000030)	MANAGER	20	F	1941-05-11	38250.00	800.00	3060.00
(000050)	MANAGER	16	M	1925-09-15	40175.00	800.00	3214.00
(000060)	MANAGER	16	M	1945-07-07	32250.00	600.00	2580.00
(000070)	MANAGER	16	F	1953-05-26	36170.00	700.00	2893.00
(000090)	MANAGER	16	F	1941-05-15	29750.00	600.00	2380.00
(000100)	MANAGER	14	M	1956-12-18	26150.00	500.00	2092.00
(000110)	SALESREP	19	M	1929-11-05	46500.00	900.00	3720.00
(000120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(000130)	ANALYST	16	F	1925-09-15	23800.00	500.00	1904.00
(000140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(000150)	DESIGNER	16	M	1947-05-17	25280.00	500.00	2022.00
(000160)	DESIGNER	17	F	1955-04-12	22250.00	400.00	1780.00
(000170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(000180)	DESIGNER	17	F	1949-02-21	21340.00	500.00	1707.00
(000190)	DESIGNER	16	M	1952-06-25	20450.00	400.00	1636.00
(000200)	DESIGNER	16	M	1941-05-29	27740.00	600.00	2217.00
(000210)	DESIGNER	17	M	1953-02-23	18270.00	400.00	1462.00
(000220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(000230)	CLERK	14	M	1935-05-30	22180.00	400.00	1774.00
(000240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00
(000250)	CLERK	15	M	1939-11-12	19180.00	400.00	1534.00
(000260)	CLERK	16	F	1936-10-05	17250.00	300.00	1380.00
(000270)	CLERK	15	F	1953-05-26	27380.00	500.00	2190.00
(000280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(000290)	OPERATOR	12	M	1946-07-09	15340.00	300.00	1227.00
(000300)	OPERATOR	14	M	1936-10-27	17750.00	400.00	1420.00
(000310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(000320)	FIELDREP	16	M	1932-08-11	19950.00	400.00	1596.00
(000330)	FIELDREP	14	M	1941-07-18	25370.00	500.00	2030.00
(000340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00
(200010)	SALESREP	18	F	1933-08-14	46500.00	1000.00	4220.00
(200120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(200140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(200170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(200220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(200240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00
(200280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(200310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(200330)	FIELDREP	14	F	1941-07-18	25370.00	500.00	2030.00
(200340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00

Employee photo and resume table (DSN8710.EMP_PHOTO_RESUME)

The employee photo and resume table complements the employee table. Each row of the photo and resume table contains a photo of the employee, in two formats, and the employee's resume. The photo and resume table resides in table space DSN8D71A.DSN8S71E. The following statement creates the table:

```
CREATE TABLE  DSN8710.EMP_PHOTO_RESUME
               (EMPNO     CHAR(06) NOT NULL,
                EMP_ROWID  ROWID NOT NULL GENERATED ALWAYS,
                PSEG_PHOTO BLOB(100K),
```

```
                    BMP_PHOTO   BLOB(100K),
                    RESUME          CLOB(5K))
                    PRIMARY KEY   EMPNO
            IN  DSN8D71L.DSN8S71B
            CCSID   EBCDIC;
```

DB2 requires an auxiliary table for each LOB column in a table. These statements
define the auxiliary tables for the three LOB columns in
DSN8710.EMP_PHOTO_RESUME:

```
CREATE AUX TABLE DSN8710.AUX_BMP_PHOTO
                IN DSN8D71L.DSN8S71M
                STORES DSN8710.EMP_PHOTO_RESUME
                COLUMN BMP_PHOTO;

CREATE AUX TABLE DSN8710.AUX_PSEG_PHOTO
                IN DSN8D71L.DSN8S71L
                STORES DSN8710.EMP_PHOTO_RESUME
                COLUMN PSEG_PHOTO;

CREATE AUX TABLE DSN8710.AUX_EMP_RESUME
                IN DSN8D71L.DSN8S71N
                STORES DSN8710.EMP_PHOTO_RESUME
                COLUMN RESUME;
```

Content

Table 113 shows the content of the columns.

Table 113. Columns of the employee photo and resume table

Column	Column Name	Description
1	EMPNO	Employee ID (the primary key)
2	EMP_ROWID	Row ID to uniquely identify each row of the table. DB2 supplies the values of this column.
3	PSEG_PHOTO	Employee photo, in PSEG format
4	BMP_PHOTO	Employee photo, in BMP format
5	RESUME	Employee resume

The employee photo and resume table has these indexes:

Table 114. Indexes of the employee photo and resume table

Name	On Column	Type of Index
DSN8710.XEMP_PHOTO_RESUME	EMPNO	Primary, ascending

The auxiliary tables for the employee photo and resume table have these indexes:

Table 115. Indexes of the auxiliary tables for the employee photo and resume table

Name	On Table	Type of Index
DSN8710.XAUX_BMP_PHOTO	DSN8710.AUX_BMP_PHOTO	Unique
DSN8710.XAUX_PSEG_PHOTO	DSN8710.AUX_PSEG_PHOTO	Unique
DSN8710.XAUX_EMP_RESUME	DSN8710.AUX_EMP_RESUME	Unique

Relationship to other tables

The table is a parent table of the project table, through a foreign key on column
RESPEMP.

Project table (DSN8710.PROJ)

The project table describes each project that the business is currently undertaking. Data contained in each row include the project number, name, person responsible, and schedule dates.

The table resides in database DSN8D71A. Because it has foreign keys referencing DEPT and EMP, those tables and the indexes on their primary keys must be created first. Then PROJ is created with:

```
CREATE TABLE DSN8710.PROJ
              (PROJNO   CHAR(6) PRIMARY KEY NOT NULL,
               PROJNAME VARCHAR(24)    NOT NULL WITH DEFAULT
                 'PROJECT NAME UNDEFINED',
               DEPTNO   CHAR(3)        NOT NULL REFERENCES
                 DSN8710.DEPT ON DELETE RESTRICT,
               RESPEMP  CHAR(6)        NOT NULL REFERENCES
                 DSN8710.EMP ON DELETE RESTRICT,
               PRSTAFF  DECIMAL(5, 2)        ,
               PRSTDATE DATE                 ,
               PRENDATE DATE                 ,
               MAJPROJ  CHAR(6))
         IN DSN8D71A.DSN8S71P
         CCSID EBCDIC;
```

Because the table is self-referencing, the foreign key for that restraint must be added later with:

```
ALTER TABLE DSN8710.PROJ
      FOREIGN KEY RPP (MAJPROJ) REFERENCES DSN8710.PROJ
              ON DELETE CASCADE;
```

Content

Table 116 shows the content of the columns.

Table 116. Columns of the project table

Column	Column Name	Description
1	PROJNO	Project ID (the primary key)
2	PROJNAME	Project name
3	DEPTNO	ID of department responsible for the project
4	RESPEMP	ID of employee responsible for the project
5	PRSTAFF	Estimated mean number of persons needed between PRSTDATE and PRENDATE to achieve the whole project, including any subprojects
6	PRSTDATE	Estimated project start date
7	PRENDATE	Estimated project end date
8	MAJPROJ	ID of any project of which this project is a part

The project table has these indexes:

Table 117. Indexes of the project table

Name	On Column	Type of Index
DSN8710.XPROJ1	PROJNO	Primary, ascending
DSN8710.XPROJ2	RESPEMP	Ascending

Relationship to other tables

The table is self-referencing: a nonnull value of MAJPROJ must be a project number. The table is a parent table of the project activity table, through a foreign key on column PROJNO. It is a dependent of:
- The department table, through its foreign key on DEPTNO
- The employee table, through its foreign key on RESPEMP.

Project activity table (DSN8710.PROJACT)

The project activity table lists the activities performed for each project. The table resides in database DSN8D71A. Because it has foreign keys referencing PROJ and ACT, those tables and the indexes on their primary keys must be created first. Then PROJACT is created with:

```
CREATE TABLE DSN8710.PROJACT
      (PROJNO    CHAR(6)                      NOT NULL,
       ACTNO     SMALLINT                     NOT NULL,
       ACSTAFF   DECIMAL(5,2)                         ,
       ACSTDATE  DATE                         NOT NULL,
       ACENDATE  DATE                                 ,
       PRIMARY KEY (PROJNO, ACTNO, ACSTDATE),
       FOREIGN KEY RPAP (PROJNO) REFERENCES DSN8710.PROJ
                                     ON DELETE RESTRICT,
       FOREIGN KEY RPAA (ACTNO) REFERENCES DSN8710.ACT
                                     ON DELETE RESTRICT)
  IN DSN8D71A.DSN8S71P
  CCSID EBCDIC;
```

Content

Table 118 shows the content of the columns.

Table 118. Columns of the project activity table

Column	Column Name	Description
1	PROJNO	Project ID
2	ACTNO	Activity ID
3	ACSTAFF	Estimated mean number of employees needed to staff the activity
4	ACSTDATE	Estimated activity start date
5	ACENDATE	Estimated activity completion date

The project activity table has this index:

Table 119. Index of the project activity table

Name	On Columns	Type of Index
DSN8710.XPROJAC1	PROJNO, ACTNO, ACSTDATE	primary, ascending

Relationship to other tables

The table is a parent table of the employee to project activity table, through a foreign key on columns PROJNO, ACTNO, and EMSTDATE. It is a dependent of:
- The activity table, through its foreign key on column ACTNO
- The project table, through its foreign key on column PROJNO

Employee to project activity table (DSN8710.EMPPROJACT)

The employee to project activity table identifies the employee who performs an activity for a project, tells the proportion of the employee's time required, and gives a schedule for the activity.

The table resides in database DSN8D71A. Because it has foreign keys referencing EMP and PROJACT, those tables and the indexes on their primary keys must be created first. Then EMPPROJACT is created with:

```
CREATE TABLE DSN8710.EMPPROJACT
     (EMPNO    CHAR(6)                          NOT NULL,
      PROJNO   CHAR(6)                          NOT NULL,
      ACTNO    SMALLINT                         NOT NULL,
      EMPTIME  DECIMAL(5,2)                             ,
      EMSTDATE DATE                                     ,
      EMENDATE DATE                                     ,
      FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
               REFERENCES DSN8710.PROJACT
                                     ON DELETE RESTRICT,
      FOREIGN KEY REPAE (EMPNO) REFERENCES DSN8710.EMP
                                     ON DELETE RESTRICT)
  IN DSN8D71A.DSN8S71P
  CCSID EBCDIC;
```

Content

Table 120 shows the content of the columns.

Table 120. Columns of the employee to project activity table

Column	Column Name	Description
1	EMPNO	Employee ID number
2	PROJNO	Project ID of the project
3	ACTNO	ID of the activity within the project
4	EMPTIME	A proportion of the employee's full time (between 0.00 and 1.00) to be spent on the activity
5	EMSTDATE	Date the activity starts
6	EMENDATE	Date the activity ends

The table has these indexes:

Table 121. Indexes of the employee to project activity table

Name	On Columns	Type of Index
DSN8710.XEMPPROJACT1	PROJNO, ACTNO, EMSTDATE, EMPNO	Unique, ascending
DSN8710.XEMPPROJACT2	EMPNO	Ascending

Relationship to other tables

The table is a dependent of:

- The employee table, through its foreign key on column EMPNO
- The project activity table, through its foreign key on columns PROJNO, ACTNO, and EMSTDATE.

Relationships among the tables

Figure 235 shows relationships among the tables. These are established by foreign keys in dependent tables that reference primary keys in parent tables. You can find descriptions of the columns with descriptions of the tables.
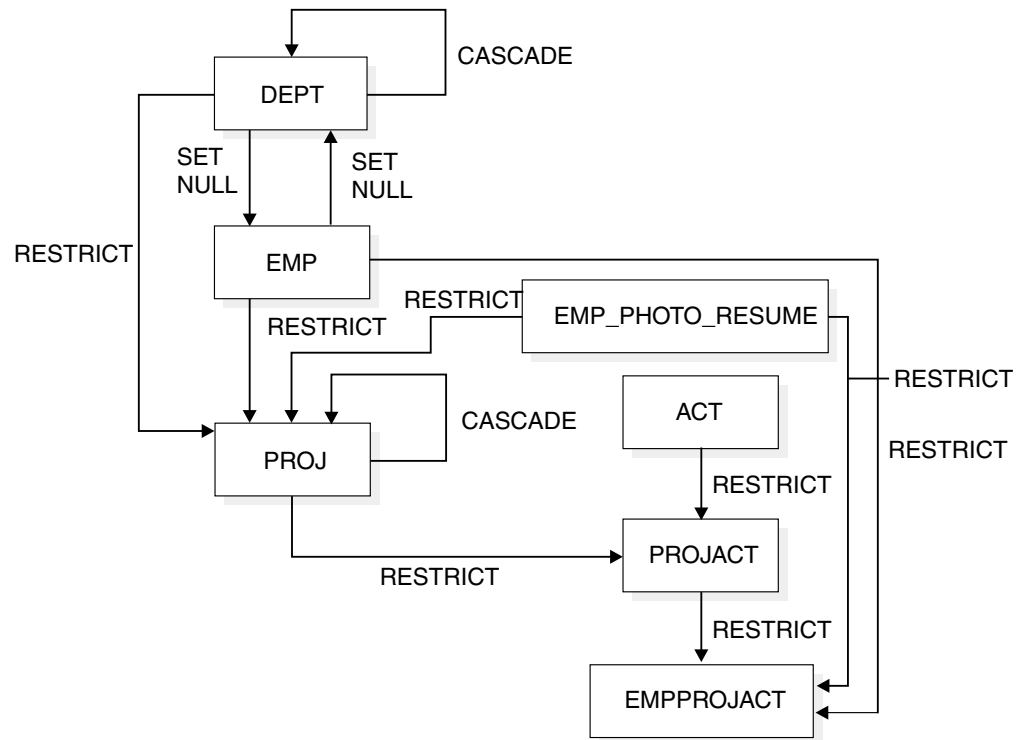

Figure 235. Relationships among tables in the sample application. Arrows point from parent tables to dependent tables.

Views on the sample tables

DB2 creates a number of views on the sample tables for use in the sample applications. Table 122 indicates the tables on which each view is defined and the sample applications that use the view. All view names have the qualifier DSN8710.

Table 122. Views on sample tables

View name	On tables or views	Used in application
VDEPT	DEPT	Organization Project
VHDEPT	DEPT	Distributed organization
VEMP	EMP	Distributed organization Organization Project
VPROJ	PROJ	Project
VACT	ACT	Project
VEMPPROJACT	EMPROJACT	Project
VDEPMG1	DEPT EMP	Organization

Table 122. Views on sample tables (continued)

View name	On tables or views	Used in application
VEMPDPT1	DEPT EMP	Organization
VASTRDE1	DEPT	
VASTRDE2	VDEPMG1 EMP	Organization
VPROJRE1	PROJ EMP	Project
VPSTRDE1	VPROJRE1 VPROJRE2	Project
VPSTRDE2	VPROJRE1	Project
VSTAFAC1	PROJACT ACT	Project
VSTAFAC2	EMPPROJACT ACT EMP	Project
VPHONE	EMP DEPT	Phone
VEMPLP	EMP	Phone

The SQL statements that create the sample views are shown below.

```
CREATE VIEW DSN8710.VDEPT
   AS SELECT ALL        DEPTNO  ,
                        DEPTNAME,
                        MGRNO   ,
                        ADMRDEPT
   FROM DSN8710.DEPT;
 CREATE VIEW DSN8710.VHDEPT
    AS SELECT ALL       DEPTNO  ,
                        DEPTNAME,
                        MGRNO   ,
                        ADMRDEPT,
                        LOCATION
     FROM DSN8710.DEPT;
CREATE VIEW DSN8710.VEMP
   AS SELECT ALL        EMPNO   ,
                        FIRSTNME,
                        MIDINIT ,
                        LASTNAME,
                        WORKDEPT
   FROM DSN8710.EMP;
CREATE VIEW DSN8710.VPROJ
   AS SELECT ALL
            PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF,
            PRSTDATE, PRENDATE, MAJPROJ
   FROM DSN8710.PROJ ;
CREATE VIEW DSN8710.VACT
     AS SELECT ALL    ACTNO   ,
                      ACTKWD  ,
                      ACTDESC
         FROM DSN8710.ACT ;
CREATE VIEW DSN8710.VPROJACT
     AS SELECT ALL
            PROJNO,ACTNO, ACSTAFF, ACSTDATE, ACENDATE
            FROM DSN8710.PROJACT ;
```

```
CREATE VIEW DSN8710.VEMPPROJACT
      AS SELECT ALL
          EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, EMENDATE
          FROM DSN8710.EMPPROJACT ;
CREATE VIEW DSN8710.VDEPMG1
      (DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT,
       LASTNAME, ADMRDEPT)
    AS SELECT ALL
       DEPTNO, DEPTNAME, EMPNO, FIRSTNME, MIDINIT,
        LASTNAME, ADMRDEPT
        FROM DSN8710.DEPT LEFT OUTER JOIN DSN8710.EMP
        ON MGRNO = EMPNO ;
CREATE VIEW DSN8710.VEMPDPT1
      (DEPTNO, DEPTNAME, EMPNO, FRSTINIT, MIDINIT,
       LASTNAME, WORKDEPT)
    AS SELECT ALL
       DEPTNO, DEPTNAME, EMPNO, SUBSTR(FIRSTNME, 1, 1), MIDINIT,
       LASTNAME, WORKDEPT
        FROM DSN8710.DEPT RIGHT OUTER JOIN DSN8710.EMP
        ON WORKDEPT = DEPTNO ;
CREATE VIEW DSN8710.VASTRDE1
    (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
     DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
    AS SELECT ALL
        D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
        D1.LASTNAME, '1',
        D2.DEPTNO,D2.DEPTNAME,D2.MGRNO,D2.FIRSTNME,D2.MIDINIT,
        D2.LASTNAME
        FROM DSN8710.VDEPMG1 D1, DSN8710.VDEPMG1 D2
        WHERE D1.DEPTNO = D2.ADMRDEPT ;
CREATE VIEW DSN8710.VASTRDE2
    (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
     DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
    AS SELECT ALL
        D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
        D1.LASTNAME,'2',
        D1.DEPTNO,D1.DEPTNAME,E2.EMPNO,E2.FIRSTNME,E2.MIDINIT,
        E2.LASTNAME
        FROM DSN8710.VDEPMG1 D1, DSN8710.EMP E2
        WHERE D1.DEPTNO = E2.WORKDEPT;
CREATE VIEW DSN8710.VPROJRE1
  (PROJNO,PROJNAME,PROJDEP,RESPEMP,FIRSTNME,MIDINIT,
   LASTNAME,MAJPROJ)
   AS SELECT ALL
      PROJNO,PROJNAME,DEPTNO,EMPNO,FIRSTNME,MIDINIT,
      LASTNAME,MAJPROJ
     FROM DSN8710.PROJ, DSN8710.EMP
    WHERE RESPEMP = EMPNO ;
CREATE VIEW DSN8710.VPSTRDE1
  (PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
   PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
   AS SELECT ALL
       P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
       P1.LASTNAME,
       P2.PROJNO,P2.PROJNAME,P2.RESPEMP,P2.FIRSTNME,P2.MIDINIT,
       P2.LASTNAME
     FROM DSN8710.VPROJRE1 P1,
       DSN8710.VPROJRE1 P2
     WHERE P1.PROJNO = P2.MAJPROJ ;
CREATE VIEW DSN8710.VPSTRDE2
  (PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
   PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
   AS SELECT ALL
       P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
```

```
                        P1.LASTNAME,
                        P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
                        P1.LASTNAME
                 FROM DSN8710.VPROJRE1 P1
                   WHERE NOT EXISTS
                      (SELECT * FROM DSN8710.VPROJRE1 P2
                         WHERE P1.PROJNO = P2.MAJPROJ) ;
CREATE VIEW DSN8710.VFORPLA
   (PROJNO,PROJNAME,RESPEMP,PROJDEP,FRSTINIT,MIDINIT,LASTNAME)
     AS SELECT ALL
        F1.PROJNO,PROJNAME,RESPEMP,PROJDEP, SUBSTR(FIRSTNME, 1, 1),
        MIDINIT, LASTNAME
      FROM DSN8710.VPROJRE1 F1 LEFT OUTER JOIN DSN8710.EMPPROJACT F2
      ON F1.PROJNO = F2.PROJNO;
CREATE VIEW DSN8710.VSTAFAC1
   (PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
    EMPTIME,STDATE,ENDATE, TYPE)
     AS SELECT ALL
          PA.PROJNO, PA.ACTNO, AC.ACTDESC,' ', ' ', ' ', ' ',
          PA.ACSTAFF, PA.ACSTDATE,
          PA.ACENDATE,'1'
        FROM DSN8710.PROJACT PA, DSN8710.ACT AC
        WHERE PA.ACTNO = AC.ACTNO ;
CREATE VIEW DSN8710.VSTAFAC2
   (PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
    EMPTIME,STDATE, ENDATE, TYPE)
     AS SELECT ALL
          EP.PROJNO, EP.ACTNO, AC.ACTDESC, EP.EMPNO,EM.FIRSTNME,
          EM.MIDINIT, EM.LASTNAME, EP.EMPTIME, EP.EMSTDATE,
          EP.EMENDATE,'2'
        FROM DSN8710.EMPPROJACT EP, DSN8710.ACT AC, DSN8710.EMP EM
        WHERE EP.ACTNO = AC.ACTNO  AND EP.EMPNO = EM.EMPNO ;

CREATE VIEW DSN8710.VPHONE
               (LASTNAME,
                FIRSTNAME,
                MIDDLEINITIAL,
                PHONENUMBER,
                EMPLOYEENUMBER,
                DEPTNUMBER,
                DEPTNAME)
     AS SELECT ALL      LASTNAME,
                        FIRSTNME,
                        MIDINIT ,
                        VALUE(PHONENO,'     '),
                        EMPNO,
                        DEPTNO,
                        DEPTNAME
     FROM DSN8710.EMP, DSN8710.DEPT
     WHERE WORKDEPT = DEPTNO;

CREATE VIEW DSN8710.VEMPLP
               (EMPLOYEENUMBER,
                PHONENUMBER)
     AS SELECT ALL      EMPNO   ,
                        PHONENO
     FROM DSN8710.EMP ;
```
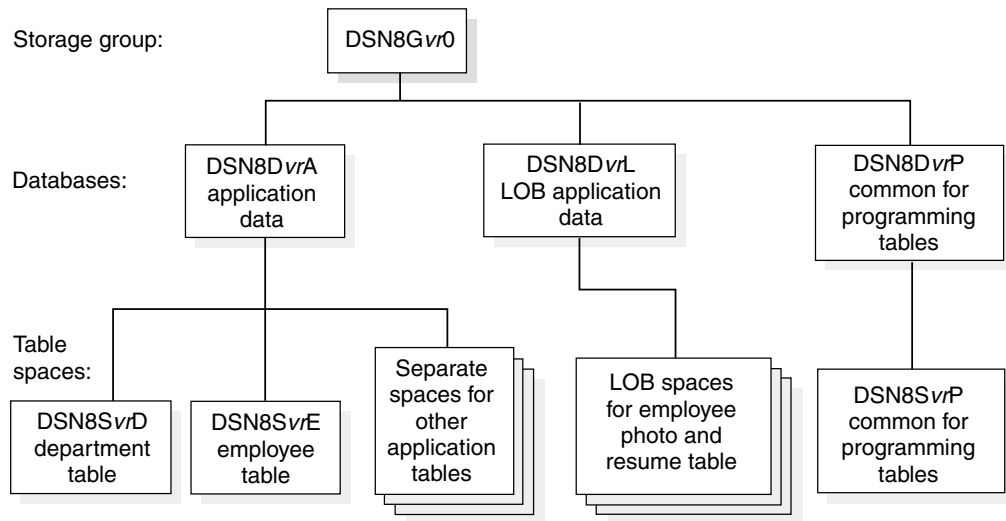
Storage of sample application tables

Figure 236 on page 825 shows how the sample tables are related to databases and storage groups. Two databases are used to illustrate the possibility. Normally, related data is stored in the same database.

vr is a 2-digit version identifer.

Figure 236. Relationship among sample databases and table spaces

In addition to the storage group and databases shown in Figure 236, the storage group DSN8G71U and database DSN8D71U are created when you run DSNTEJ2A.

Storage group

The default storage group, SYSDEFLT, created when DB2 is installed, is not used to store sample application data. The storage group used to store sample application data is defined by this statement:

```
CREATE STOGROUP DSN8G710
  VOLUMES (DSNV01)
  VCAT  DSNC710;
```

Databases

The default database, created when DB2 is installed, is not used to store the sample application data. Two databases are used: one for tables related to applications, the other for tables related to programs. They are defined by the following statements:

```
CREATE DATABASE DSN8D71A
  STOGROUP DSN8G710
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D71P
  STOGROUP DSN8G710
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D71L
  STOGROUP DSN8G710
  BUFFERPOOL BP0
  CCSID EBCDIC;
```

Table spaces

The following table spaces are explicitly defined by the statements shown below. The table spaces not explicitly defined are created implicitly in the DSN8D71A database, using the default space attributes.

```
CREATE TABLESPACE DSN8S71D
  IN DSN8D71A
  USING STOGROUP DSN8G710
        PRIQTY 20
        SECQTY 20
        ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S71E
  IN DSN8D71A
  USING STOGROUP DSN8G710
        PRIQTY 20
        SECQTY 20
        ERASE NO
  NUMPARTS 4
    (PART 1 USING STOGROUP DSN8G710
                   PRIQTY 12
                   SECQTY 12,
     PART 3 USING STOGROUP DSN8G710
                   PRIQTY 12
                   SECQTY 12)
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  COMPRESS YES
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S71B
  IN DSN8D71L
  USING STOGROUP DSN8G710
        PRIQTY 20
        SECQTY 20
        ERASE NO
  LOCKSIZE PAGE
  LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE LOB TABLESPACE DSN8S71M
  IN DSN8D71L
  LOG NO;

CREATE LOB TABLESPACE DSN8S71L
  IN DSN8D71L
  LOG NO;

CREATE LOB TABLESPACE DSN8S71N
  IN DSN8D71L
  LOG NO;
```

```
CREATE TABLESPACE DSN8S71C
  IN DSN8D71P
  USING STOGROUP DSN8G710
        PRIQTY 160
        SECQTY 80
  SEGSIZE 4
  LOCKSIZE TABLE
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S71P
  IN DSN8D71A
  USING STOGROUP DSN8G710
        PRIQTY 160
        SECQTY 80
  SEGSIZE 4
  LOCKSIZE ROW
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;
CREATE TABLESPACE DSN8S71R
  IN DSN8D71A
  USING STOGROUP DSN8G710
        PRIQTY 20
        SECQTY 20
        ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;
CREATE TABLESPACE DSN8S71S
   IN DSN8D71A
   USING STOGROUP DSN8G710
             PRIQTY 20
             SECQTY 20
             ERASE NO
   LOCKSIZE PAGE LOCKMAX SYSTEM
   BUFFERPOOL BP0
   CLOSE NO
   CCSID EBCDIC;
```

Appendix B. Sample applications

This appendix describes the DB2 sample applications and the environments under which each application runs. It also provides information on how to use the applications, and how to print the application listings.

Several sample applications come with DB2 to help you with DB2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing to company.

You can examine the source code for the sample application programs in the online sample library included with the DB2 product. The name of this sample library is *prefix*.SDSNSAMP.

Types of sample applications

Organization application: The organization application manages the following company information:
- Department administrative structure
- Individual departments
- Individual employees.

Management of information about department administrative structures involves how departments relate to other departments. You can view or change the organizational structure of an individual department, and the information about individual employees in any department. The organization application runs interactively in the ISPF/TSO, IMS, and CICS environments and is available in PL/I and COBOL.

Project application: The project application manages information about a company's project activities, including the following:
- Project structures
- Project activity listings
- Individual project processing
- Individual project activity estimate processing
- Individual project staffing processing.

Each department works on projects that contain sets of related activities. Information available about these activities includes staffing assignments, completion-time estimates for the project as a whole, and individual activities within a project. The project application runs interactively in IMS and CICS and is available in PL/I only.

Phone application: The phone application lets you view or update individual employee phone numbers. There are different versions of the application for ISPF/TSO, CICS, IMS, and batch:
- ISPF/TSO applications use COBOL and PL/I.
- CICS and IMS applications use PL/I.
- Batch applications use C, C⁺⁺, COBOL, FORTRAN, and PL/I.

Stored procedure applications: There are three sets of stored procedure applications:
- IFI applications

These applications let you pass DB2 commands from a client program to a stored procedure, which runs the commands at a DB2 server using the instrumentation facility interface (IFI). There are two sets of client programs and stored procedures. One set has a PL/I client and stored procedure; the other set has a C client and stored procedure.

- ODBA application

  This application demonstrates how you can use the IMS ODBA interface to access IMS databases from stored procedures. The stored procedure accesses the IMS sample DL/I database. The client program and the stored procedure are written in COBOL.

- Utilities stored procedure application

  This application demonstrates how to call the utilities stored procedure. For more information on the utilities stored procedure, see Appendix B of *DB2 Utility Guide and Reference*.

- SQL procedure applications

  These applications demonstrate how to write, prepare, and invoke SQL procedures. One set of applications demonstrates how to prepare SQL procedures using JCL. The other set of applications shows how to prepare SQL procedures using the SQL procedure processor. The client programs are written in C.

- WLM refresh application

  This application is a client program that calls the DB2–supplied stored procedure WLM_REFRESH to refresh a WLM environment. This program is written in C.

- System parameter reporting application

  This application is a client program that calls the DB2–supplied stored procedure DSNWZP to display the current settings of system parameters. This program is written in C.is a stored procedure and a client program that can be used to refresh a WLM environment. This program is written in C.

All stored procedure applications run in the TSO batch environment.

User-Defined Function applications: The user-defined function applications consist of a client program that invokes the sample user-defined functions and a set of user-defined functions that perform the following functions:
- Convert the current date to a user-specified format
- Convert a date from one format to another
- Convert the current time to a user-specified format
- Convert a date from one format to another
- Return the day of the week for a user-specified date
- Return the month for a user-specified date
- Format a floating point number as a currency value
- Return the table name for a table, view, or alias
- Return the qualifier for a table, view or alias
- Return the location for a table, view or alias
- Return a table of weather information

All programs are written in C or C++ and run in the TSO batch environment.

LOB application: The LOB application demonstrates how to perform the following tasks:
- Define DB2 objects to hold LOB data
- Populate DB2 tables with LOB data using the LOAD utility, or using INSERT and UPDATE statements when the data is too large for use with the LOAD utility
- Manipulate the LOB data using LOB locators

The programs that create and populate the LOB objects use DSNTIAD and run in the TSO batch environment. The program that manipulates the LOB data is written in C and runs under ISPF/TSO.

Using the applications

You can use the applications interactively by accessing data in the sample tables on screen displays (panels). You can also access the sample tables in batch when using the phone applications. Part 2 of *DB2 Installation Guide* contains detailed information about using each application. All sample objects have PUBLIC authorization, which makes the samples easier to run.

Application languages and environments: Table 123 shows the environments under which each application runs, and the languages the applications use for each environment.

Table 123. Application languages and environments

Programs	ISPF/TSO	IMS	CICS	Batch	SPUFI
Dynamic SQL Programs				Assembler PL/I	
Exit Routines	Assembler	Assembler	Assembler	Assembler	Assembler
Organization	COBOL[1]	COBOL PL/I	COBOL PL/I		
Phone	COBOL PL/I Assembler[2]	PL/I	PL/I	COBOL FORTRAN PL/I C C++	
Project		PL/I	PL/I		
SQLCA Formatting Routines		Assembler	Assembler	Assembler	Assembler
Stored Procedures		COBOL		PL/I C SQL	
User-Defined Functions				C C++	
LOBs	C				

Notes:

1. For all instances of COBOL in this table, the application can be compiled using OS/VS COBOL, VS/COBOL II, or IBM COBOL for MVS & VM.

2. Assembler subroutine DSN8CA.

Application programs: Tables 124 through 126 on pages 832 through 834 provide the program names, JCL member names, and a brief description of some of the programs included for each of the three environments: TSO, IMS, and CICS.

TSO

Table 124. Sample DB2 applications for TSO

Application	Program name	Preparation JCL member name	Attachment facility	Description
Phone	DSN8BC3	DSNTEJ2C	DSNELI	This COBOL batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BD3	DSNTEJ2D	DSNELI	This C batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BE3	DSNTEJ2E	DSNELI	This C++ batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BP3	DSNTEJ2P	DSNELI	This PL/I batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BF3	DSNTEJ2F	DSNELI	This FORTRAN program lists employee telephone numbers and updates them if requested.
Organization	DSN8HC3	DSNTEJ3C DSNTEJ6	DSNALI	This COBOL ISPF program displays and updates information about a local department. It can also display and update information about an employee at a local or remote location.
Phone	DSN8SC3	DSNTEJ3C	DSNALI	This COBOL ISPF program lists employee telephone numbers and updates them if requested.
Phone	DSN8SP3	DSNTEJ3P	DSNALI	This PL/I ISPF program lists employee telephone numbers and updates them if requested.
UNLOAD	DSNTIAUL	DSNTEJ2A	DSNELI	This assembler language program allows you to unload the data from a table or view and to produce LOAD utility control statements for the data.
Dynamic SQL	DSNTIAD	DSNTIJTM	DSNELI	This assembler language program dynamically executes non-SELECT statements read in from SYSIN; that is, it uses dynamic SQL to execute non-SELECT SQL statements.
Dynamic SQL	DSNTEP2	DSNTEJ1P or DSNTEJ1L	DSNELI	This PL/I program dynamically executes SQL statements read in from SYSIN. Unlike DSNTIAD, this application can also execute SELECT statements.

Application	Program name	Preparation JCL member name	Attachment facility	Description
Stored Procedures	DSN8EP1 DSN8EP2 DSN8EPU DSN8ED1 DSN8ED2 DSN8EC1 DSN8EC2 DSN8ES1 DSN8ED3 DSN8ES2 DSN8ED6 DSN8ED7	DSNTEJ6P DSNTEJ6S DSNTEJ6U DSNTEJ6D DSNTEJ6T DSNTEJ61 DSNTEJ62 DSNTEJ63 DSNTEJ64 DSNTEJ65 DSNTEJ6W DSNTEJ6Z	DSNELI DSNALI DSNELI DSNELI DSNALI DSNRLI DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI	These applications consist of a calling program, a stored procedure program, or both. Samples that are prepared by jobs DSNTEJ6P, DSNTEJ6S, DSNTEJ6D, and DSNTEJ6T execute DB2 commands using the instrumentation facility interface (IFI). DSNTEJ6P and DSNTEJ6S prepare a PL/I version of the application. DSNTEJ6D and DSNTEJ6T prepare a version in C. The C stored procedure uses result sets to return commands to the client. The sample that is prepared by DSNTEJ61 and DSNTEJ62 demonstrates a stored procedure that accesses IMS databases through the ODBA interface. The sample that is prepared by DSNTEJ6U invokes the utilities stored procedure. The sample that is prepared by jobs DSNTEJ63 and DSNTEJ64 demonstrates how to prepare an SQL procedure using JCL. The sample that is prepared by job DSNTEJ65 demonstrates how to prepare an SQL procedure using the SQL procedure processor. The sample that is prepared by job DSNTEJ6W demonstrates how to prepare and run a client program that calls a DB2–supplied stored procedure to refresh a WLM environment. The sample that is prepared by job DSNTEJ6Z demonstrates how to prepare and run a client program that calls a DB2–supplied stored procedure to display the current settings of system parameters.
User-Defined Functions	DSN8DUAD DSN8DUAT DSN8DUCD DSN8DUCT DSN8DUCY DSN8DUTI DSN8DUWC DSN8DUWF DSN8EUDN DSN8EUMN	DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U DSNTEJ2U	DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI DSNELI	These applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2 and one user-defined table function, DSN8DUWF, that can be invoked by client program DSN8DUWC. DSN8EUDN and DSN8EUMN are written in C++. All other programs are written in C.
LOBs	DSN8DLPL DSN8DLCT DSN8DLRV DSN8DLPV	DSNTEJ71 DSNTEJ71 DSNTEJ73 DSNTEJ75	DSNELI DSNELI DSNELI DSNELI	These applications demonstrate how to populate a LOB column that is greater than 32KB, manipulate the data using the POSSTR and SUBSTR built-in functions, and display the data in ISPF using GDDM.

IMS

Table 125. Sample DB2 applications for IMS

Application	Program name	JCL member name	Description
Organization	DSN8IC0 DSN8IC1 DSN8IC2	DSNTEJ4C	IMS COBOL Organization Application
Organization	DSN8IP0 DSN8IP1 DSN8IP2	DSNTEJ4P	IMS PL/I Organization Application
Project	DSN8IP6 DSN8IP7 DSN8IP8	DSNTEJ4P	IMS PL/I Project Application
Phone	DSN8IP3	DSNTEJ4P	IMS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

CICS

Table 126. Sample DB2 applications for CICS

Application	Program name	JCL member name	Description
Organization	DSN8CC0 DSN8CC1 DSN8CC2	DSNTEJ5C	CICS COBOL Organization Application
Organization	DSN8CP0 DSN8CP1 DSN8CP2	DSNTEJ5P	CICS PL/I Organization Application
Project	DSN8CP6 DSN8CP7 DSN8CP8	DSNTEJ5P	CICS PL/I Project Application
Phone	DSN8CP3	DSNTEJ5P	CICS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Appendix C. How to run sample programs DSNTIAUL, DSNTIAD, and DSNTEP2

DB2 provides three sample programs that many users find helpful as productivity aids. These programs are shipped as source code, so you can modify them to meet your needs. The programs are:

DSNTIAUL   The sample unload program. This program, which is written in assembler language, unloads some or all rows from up to 100 DB2 tables. With DSNTIAUL, you can unload data of any DB2 built-in data type or distinct type. You can unload up to 32KB of data from a LOB column. DSNTIAUL unloads the rows in a form that is compatible with the LOAD utility and generates utility control statements for LOAD. DSNTIAUL also lets you execute any SQL non-SELECT statement that can be executed dynamically.

DSNTIAD   A sample dynamic SQL program in assembler language. With this program, you can execute any SQL statement that can be executed dynamically, except a SELECT statement.

DSNTEP2   A sample dynamic SQL program in the PL/I language. With this program, you can execute any SQL statement that can be executed dynamically. You can use the source version of DSNTEP2 and modify it to meet your needs, or, if you do not have a PL/I compiler at your installation, you can use the object code version of DSNTEP2.

Because these three programs also accept the static SQL statements CONNECT, SET CONNECTION, and RELEASE, you can use the programs to access DB2 tables at remote locations.

DSNTIAUL and DSNTIAD are shipped only as source code, so you must precompile, assemble, link, and bind them before you can use them. If you want to use the source code version of DSNTEP2, you must precompile, compile, link and bind it. You need to bind the object code version of DSNTEP2 before you can use it. Usually, your system administrator prepares the programs as part of the installation process. Table 127 indicates which installation job prepares each sample program. All installation jobs are in data set DSN710.SDSNSAMP.

Table 127. Jobs that prepare DSNTIAUL, DSNTIAD, and DSNTEP2

Program name	Program preparation job
DSNTIAUL	DSNTEJ2A
DSNTIAD	DSNTIJTM
DSNTEP2 (source)	DSNTEJ1P
DSNTEP2 (object)	DSNTEJ1L

To run the sample programs, use the DSN RUN command, which is described in detail in Chapter 2 of *DB2 Command Reference*. Table 128 on page 836 lists the load module name and plan name you must specify, and the parameters you can specify when you run each program. See the following sections for the meaning of each parameter.

Table 128. DSN RUN option values for DSNTIAUL, DSNTIAD, and DSNTEP2

Program name	Load module	Plan	Parameters
DSNTIAUL	DSNTIAUL	DSNTIB71	SQL
DSNTIAD	DSNTIAD	DSNTIA71	RC0 SQLTERM(*termchar*)
DSNTEP2	DSNTEP2	DSNTEP71	ALIGN(MID) or ALIGN(LHS) NOMIXED or MIXED SQLTERM(*termchar*)

The remainder of this appendix contains the following information about running each program:
- Descriptions of the input parameters
- Data sets you must allocate before you run the program
- Return codes from the program
- Examples of invocation

See the sample jobs listed in Table 127 on page 835 for a working example of each program.

Running DSNTIAUL

This section contains information that you need when you run DSNTIAUL, including parameters, data sets, return codes, and invocation examples.

DSNTIAUL parameters: DSNTIAUL accepts one parameter, SQL. If you specify this parameter, your input data set contains one or more complete SQL statements, each of which ends with a semi-colon. You can include any SQL statement that can be executed dynamically in your input data set. In addition, you can include the static SQL statements CONNECT, SET CONNECTION, or RELEASE. The maximum length for a statement is 32765 bytes. DSNTIAUL uses the SELECT statements to determine which tables to unload and dynamically executes all other statements except CONNECT, SET CONNECTION, and RELEASE. DSNTIAUL executes CONNECT, SET CONNECTION, and RELEASE statically to connect to remote locations.

If you do not specify the SQL parameter, your input data set must contain one or more single-line statements (without a semi-colon) that use the following syntax:

```
table or view name [WHERE conditions] [ORDER BY columns]
```

Each input statement must be a valid SQL SELECT statement with the clause SELECT * FROM omitted and with no ending semi-colon. DSNTIAUL generates a SELECT statement for each input statement by appending your input line to SELECT * FROM, then uses the result to determine which tables to unload. For this input format, the text for each table specification can be a maximum of 72 bytes and must not span multiple lines.

For both input formats, you can specify SELECT statements that join two or more tables or select specific columns from a table. If you specify columns, you will need to modify the LOAD statement that DSNTIAUL generates.

DSNTIAUL data sets:

Data set   **Description**

SYSIN   Input data set. See *DSNTIAUL parameters* for information on the contents of the input data.

You cannot enter comments in DSNTIAUL input.

The record length for the input data set must be at least 72 bytes. DSNTIAUL reads only the first 72 bytes of each record.

SYSPRINT   Output data set. DSNTIAUL writes informational and error messages in this data set.

The record length for the SYSPRINT data set is 121 bytes.

SYSPUNCH   Output data set. DSNTIAUL writes the LOAD utility control statements in this data set.

SYSREC*nn*   Output data sets. The value *nn* ranges from 00 to 99. You can have a maximum of 100 output data sets for a single execution of DSNTIAUL. Each data set contains the data unloaded when DSNTIAUL processes a SELECT statement from the input data set. Therefore, the number of output data sets must match the number of SELECT statements (if you specify parameter SQL) or table specifications in your input data set.

Define all data sets as sequential data sets. You can specify the record length and block size of the SYSPUNCH and SYSREC*nn* data sets. The maximum record length for the SYSPUNCH and SYSREC*nn* data sets is 32760 bytes.

DSNTIAUL return codes:

Return code	Meaning
0	Successful completion.
4	An SQL statement received a warning code. If the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation.
8	An SQL statement received an error code. If the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation.
12	DSNTIAUL could not open a data set, an SQL statement returned a severe error code (-8*nn* or -9*nn*), or an error occurred in the SQL message formatting routine.

Examples of DSNTIAUL invocation: Suppose you want to unload the rows for department D01 from the project table. You can fit the table specification on one line, and you do not want to execute any non-SELECT statements, so you do not need the SQL parameter. Your invocation looks like this:

```
//UNLOAD  EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB71) -
      LIB('DSN710.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,
//         UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
//         VOL=SER=SCR03
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
//         UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
//         VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN    DD *
DSN8710.PROJ WHERE DEPTNO='D01'
```

Figure 237. DSNTIAUL Invocation without the SQL parameter

If you want to obtain the LOAD utility control statements for loading rows into a
table, but you do not want to unload the rows, you can set the data set names for
the SYSREC*nn* data sets to DUMMY. For example, to obtain the utility control
statements for loading rows into the department table, you invoke DSNTIAUL like
this:

```
//UNLOAD  EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB71) -
      LIB('DSN710.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DUMMY
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
//         UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
//         VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN    DD *
DSN8710.DEPT
```

Figure 238. DSNTIAUL Invocation to obtain LOAD control statements

Now suppose that you also want to use DSNTIAUL to do these things:
- Unload all rows from the project table
- Unload only rows from the employee table for employees in departments with
  department numbers that begin with D, and order the unloaded rows by
  employee number
- Lock both tables in share mode before you unload them

For these activities, you must specify the SQL parameter when you run DSNTIAUL.
Your DSNTIAUL invocation looks like this:

```
//UNLOAD   EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB71) PARMS('SQL') -
      LIB('DSN710.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,
//            UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
//            VOL=SER=SCR03
//SYSREC01 DD DSN=DSN8UNLD.SYSREC01,
//            UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
//            VOL=SER=SCR03
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
//            UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
//            VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN    DD *
LOCK TABLE DSN8710.EMP IN SHARE MODE;
LOCK TABLE DSN8710.PROJ IN SHARE MODE;
SELECT * FROM DSN8710.PROJ;
SELECT * FROM DSN8710.EMP
  WHERE WORKDEPT LIKE 'D%'
  ORDER BY EMPNO;
```

Figure 239. DSNTIAUL Invocation with the SQL parameter

Running DSNTIAD

This section contains information that you need when you run DSNTIAD, including parameters, data sets, return codes, and invocation examples.

DSNTIAD parameters:

RC0

If you specify this parameter, DSNTIAD ends with return code 0, even if the program encounters SQL errors. If you do not specify RC0, DSNTIAD ends with a return code that reflects the severity of the errors that occur. Without RC0, DSNTIAD terminates if more than 10 SQL errors occur during a single execution.

SQLTERM(_termchar_**)**

Specify this parameter to indicate the character that you use to end each SQL statement. You can use any special character *except* one of those listed in Table 129. SQLTERM(;) is the default.

Table 129. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'5E'
double quote	"	X'7F'
left parenthesis	(	X'4D'
right parenthesis	)	X'5D'
single quote	'	X'7D'
underscore	_	X'6D'

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

Be careful to choose a character for the statement terminator that is not used within the statement.

DSNTIAD data sets:

Data Set	Description
SYSIN	Input data set. In this data set, you can enter any number of non-SELECT SQL statements, each terminated with a semi-colon. A statement can span multiple lines, but DSNTIAD reads only the first 72 bytes of each line.
	You cannot enter comments in DSNTIAD input.
SYSPRINT	Output data set. DSNTIAD writes informational and error messages in this data set. DSNTIAD sets the record length of this data set to 121 and the block size to 1210.

Define all data sets as sequential data sets.

DSNTIAD return codes:

Return code	Meaning
0	Successful completion, or the user specified parameter RC0.
4	An SQL statement received a warning code.
8	An SQL statement received an error code.
12	DSNTIAD could not open a data set, the length of an SQL statement was more than 32 760 bytes, an SQL statement returned a severe error code (-8nn or -9nn), or an error occurred in the SQL message formatting routine.

Example of DSNTIAD invocation: Suppose you want to execute 20 UPDATE statements, and you do not want DSNTIAD to terminate if more than 10 errors occur. Your invocation looks like this:

```
//RUNTIAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAD) PLAN(DSNTIA71) PARMS('RC0') -
      LIB('DSN710.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN    DD *
UPDATE DSN8710.PROJ SET DEPTNO='J01' WHERE DEPTNO='A01';
UPDATE DSN8710.PROJ SET DEPTNO='J02' WHERE DEPTNO='A02';
:
UPDATE DSN8710.PROJ SET DEPTNO='J20' WHERE DEPTNO='A20';
```

Figure 240. DSNTIAD Invocation with the RC0 Parameter

Running DSNTEP2

This section contains information that you need when you run DSNTEP2, including parameters, data sets, return codes, and invocation examples.

DSNTEP2 parameters:

Parameter
 Description

ALIGN(MID) or ALIGN(LHS)
 If you want your DSNTEP2 output centered, specify ALIGN(MID). If you want the output left-aligned, choose ALIGN(LHS). The default is ALIGN(MID).

NOMIXED or MIXED
 If your input to DSNTEP2 contains any DBCS characters, specify MIXED. If your input contains no DBCS characters, specify NOMIXED. The default is NOMIXED.

SQLTERM(*termchar***)**
 Specify this parameter to indicate the character that you use to end each SQL statement. You can use any character *except* one of those listed in Table 129 on page 839. SQLTERM(;) is the default.

 Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:
```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

 Be careful to choose a character for the statement terminator that is not used within the statement.

 If you want to change the SQL terminator within a series of SQL statements, you can use the --#SET TERMINATOR control statement. For example, suppose that you have an existing set of SQL statements to which you want to add a CREATE TRIGGER statement that has embedded semicolons. You can use the default SQLTERM value, which is a semicolon, for all of the existing

SQL statements. Before you execute the CREATE TRIGGER statement, include the --#SET TERMINATOR # control statement to change the SQL terminator to the character #:

```
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJACT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;
--#SET TERMINATOR #
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

See the discussion of the SYSIN data set for more information on the --#SET control statement.

DSNTEP2 data sets:

Data Set	Description
SYSIN	Input data set. In this data set, you can enter any number of SQL statements, each terminated with a semi-colon. A statement can span multiple lines, but DSNTEP2 reads only the first 72 bytes of each line.

You can enter comments in DSNTEP2 input with an asterisk (*) in column 1 or two hyphens (--) anywhere on a line. Text that follows the asterisk is considered to be comment text. Text that follows two hyphens can be comment text or a control statement. Comments and control statements cannot span lines.

You can enter a number of control statements in the DSNTEP2 input data set. Those control statements are of the form

```
--#SET control-option value
```

The control options are:

TERMINATOR
The SQL statement terminator. *value* is any single-byte character other than one of those listed in Table 129 on page 839. The default is the value of the SQLTERM parameter.

ROWS_FETCH
The number of rows to be fetched from the result table. *value* is a numeric literal between -1 and the number of rows in the result table. -1 means that all rows are to be fetched. The default is -1.

ROWS_OUT
The number of fetched rows to be sent to the output data set. *value* is a numeric literal between -1 and the number of fetched rows. -1 means that all fetched rows are to be sent to the output data set. The default is -1.

SYSPRINT	Output data set. DSNTEP2 writes informational and error messages in this data set. DSNTEP2 writes output records of no more than 133 bytes.

Define all data sets as sequential data sets.

DSNTEP2 return codes:

Return code	Meaning
0	Successful completion.
4	An SQL statement received a warning code.
8	An SQL statement received an error code.
12	The length of an SQL statement was more than 32 760 bytes, an SQL statement returned a severe error code (-8*nn* or -9*nn*), or an error occurred in the SQL message formatting routine.

Example of DSNTEP2 invocation: Suppose you want to use DSNTEP2 to execute SQL SELECT statements that might contain DBCS characters. You also want your output left-aligned. Your invocation looks like this:

```
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTEP2) PLAN(DSNTEP71) PARMS('/ALIGN(LHS) MIXED') -
      LIB('DSN710.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN    DD *
SELECT * FROM DSN8710.PROJ;
```

Figure 241. DSNTEP2 Invocation with the ALIGN(LHS) and MIXED parameters

Appendix D. Programming examples

This appendix contains the following programming examples:
- Sample COBOL dynamic SQL program
- "Sample dynamic and static SQL in a C program" on page 859
- "Example DB2 REXX application" on page 862
- "Sample COBOL program using DRDA access" on page 876
- "Sample COBOL program using DB2 private protocol access" on page 884
- "Examples of using stored procedures" on page 890

To prepare and run these applications, use the JCL in DSN710.SDSNSAMP as a model for your JCL. See "Appendix B. Sample applications" on page 829 for a list JCL procedures for preparing sample programs. See Part 2 of *DB2 Installation Guide* for information on the appropriate compiler options to use for each language.

Sample COBOL dynamic SQL program

"Chapter 23. Coding dynamic SQL in application programs" on page 495 describes three variations of dynamic SQL statements:

- Non-SELECT statements
- Fixed-List SELECT statements

  In this case, you know the number of columns returned and their data types when you write the program.

- Varying-List SELECT statements.

  In this case, you do **not** know the number of columns returned and their data types when you write the program.

This appendix documents a technique of coding varying list SELECT statements in VS COBOL II, COBOL/370, or IBM COBOL for MVS & VM. In the rest of this appendix, *COBOL* refers to those versions only.

This example program does not support BLOB, CLOB, or DBCLOB data types.

Pointers and based variables

COBOL has a POINTER type and a SET statement that provide pointers and based variables.

The SET statement sets a pointer from the address of an area in the linkage section or another pointer; the statement can also set the address of an area in the linkage section. Figure 243 on page 849 provides these uses of the SET statement. The SET statement does not permit the use of an address in the WORKING-STORAGE section.

Storage allocation

COBOL does not provide a means to allocate main storage within a program. You can achieve the same end by having an initial program which allocates the storage, and then calls a second program that manipulates the pointer. (COBOL does not permit you to directly manipulate the pointer because errors and abends are likely to occur.)

The initial program is extremely simple. It includes a working storage section that allocates the maximum amount of storage needed. This program then calls the

845

second program, passing the area or areas on the CALL statement. The second program defines the area in the linkage section and can then use pointers within the area.

If you need to allocate parts of storage, the best method is to use indexes or subscripts. You can use subscripts for arithmetic and comparison operations.

Example

Figure 242 on page 847 shows an example of the initial program DSN8BCU1 that allocates the storage and calls the second program DSN8BCU2 shown in Figure 243 on page 849. DSN8BCU2 then defines the passed storage areas in its linkage section and includes the USING clause on its PROCEDURE DIVISION statement.

Defining the pointers, then redefining them as numeric, permits some manipulation of the pointers that you cannot perform directly. For example, you cannot add the column length to the record pointer, but you can add the column length to the numeric value that redefines the pointer.

```
**** DSN8BCU1- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM  ***********
*                                                              *
*   MODULE NAME = DSN8BCU1                                      *
*                                                              *
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                  *
*                      UNLOAD PROGRAM                           *
*                      BATCH                                    *
*                      VS COBOL II, COBOL/370, OR               *
*                      IBM COBOL FOR MVS & VM                   *
*                                                              *
*   FUNCTION = THIS MODULE PROVIDES THE STORAGE NEEDED BY       *
*              DSN8BCU2 AND CALLS THAT PROGRAM.                 *
*                                                              *
*   NOTES =                                                     *
*     DEPENDENCIES = VS COBOL II IS REQUIRED.  SEVERAL NEW      *
*                    FACILITIES ARE USED.                       *
*                                                              *
*     RESTRICTIONS =                                            *
*              THE MAXIMUM NUMBER OF COLUMNS IS 750,            *
*              WHICH IS THE SQL LIMIT.                          *
*                                                              *
*              DATA RECORDS ARE LIMITED TO 32700 BYTES,         *
*              INCLUDING DATA, LENGTHS FOR VARCHAR DATA,        *
*              AND SPACE FOR NULL INDICATORS.                   *
*                                                              *
*   MODULE TYPE = COBOL PROGRAM                                 *
*       PROCESSOR   = VS COBOL II, COBOL/370 OR                 *
*                     IBM COBOL FOR MVS & VM                    *
*       MODULE SIZE = SEE LINK EDIT                             *
*       ATTRIBUTES  = REENTRANT                                 *
*                                                              *
*   ENTRY POINT = DSN8BCU1                                      *
*       PURPOSE = SEE FUNCTION                                  *
*       LINKAGE = INVOKED FROM DSN RUN                          *
*       INPUT   = NONE                                          *
*       OUTPUT  = NONE                                          *
*                                                              *
*   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION               *
*                                                              *
*   EXIT-ERROR =                                                *
*       RETURN CODE = NONE                                      *
*       ABEND CODES =  NONE                                     *
*       ERROR-MESSAGES = NONE                                   *
*                                                              *
*   EXTERNAL REFERENCES =                                       *
*       ROUTINES/SERVICES =                                     *
*            DSN8BCU2 - ACTUAL UNLOAD PROGRAM                   *
*                                                              *
*       DATA-AREAS        =    NONE                             *
*       CONTROL-BLOCKS    =    NONE                             *
*                                                              *
*   TABLES = NONE                                               *
*   CHANGE-ACTIVITY = NONE                                      *
```

Figure 242. Initial program that allocates storage (Part 1 of 2)

```
*                                                                    *
*  *PSEUDOCODE*                                                      *
*                                                                    *
*    PROCEDURE                                                       *
*    CALL DSN8BCU2.                                                  *
*    END.                                                            *
*--------------------------------------------------------------------*
/
 IDENTIFICATION DIVISION.
*-----------------------
 PROGRAM-ID.    DSN8BCU1
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 DATA DIVISION.
*
 WORKING-STORAGE SECTION.
*
 01  WORKAREA-IND.
         02  WORKIND PIC S9(4) COMP OCCURS 750 TIMES.
 01  RECWORK.
         02  RECWORK-LEN PIC S9(8) COMP VALUE 32700.
         02  RECWORK-CHAR PIC X(1) OCCURS 32700 TIMES.
*
 PROCEDURE DIVISION.
*
         CALL 'DSN8BCU2' USING WORKAREA-IND RECWORK.
         GOBACK.
```

Figure 242. Initial program that allocates storage (Part 2 of 2)

```
                **** DSN8BCU2- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM  ***********
                *                                                                *
                *    MODULE NAME = DSN8BCU2                                       *
                *                                                                *
                *    DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                   *
                *                       UNLOAD PROGRAM                            *
                *                       BATCH                                     *
                *                       VS COBOL II, COBOL/370, OR                *
                *                       IBM COBOL FOR MVS & VM                    *
                *                                                                *
                *    FUNCTION = THIS MODULE ACCEPTS A TABLE NAME OR VIEW NAME     *
                *              AND UNLOADS THE DATA IN THAT TABLE OR VIEW.        *
                *    READ IN A TABLE NAME FROM SYSIN.                            *
                *    PUT DATA FROM THE TABLE INTO DD SYSREC01.                   *
                *    WRITE RESULTS TO SYSPRINT.                                  *
                *                                                                *
                *    NOTES =                                                      *
                *      DEPENDENCIES = CANNOT USE OS/VS COBOL.                     *
                *                                                                *
                *      RESTRICTIONS =                                            *
                *                THE SQLDA IS LIMITED TO 33016 BYTES.            *
                *                THIS SIZE ALLOWS FOR THE DB2 MAXIMUM            *
                *                OF 750 COLUMNS.                                  *
                *                                                                *
                *                DATA RECORDS ARE LIMITED TO 32700 BYTES,        *
                *                INCLUDING DATA, LENGTHS FOR VARCHAR DATA,        *
                *                AND SPACE FOR NULL INDICATORS.                   *
                *                                                                *
                *                TABLE OR VIEW NAMES ARE ACCEPTED, AND ONLY       *
                *                ONE NAME IS ALLOWED PER RUN.                     *
                *                                                                *
                *    MODULE TYPE = COBOL PROGRAM                                  *
                *       PROCESSOR  = DB2  PRECOMPILER                            *
                *                    VS/COBOL II, COBOL/370, OR                   *
                *                    IBM COBOL FOR MVS & VM                       *
                *       MODULE SIZE = SEE LINK EDIT                              *
                *       ATTRIBUTES  = REENTRANT                                  *
                *                                                                *
                *    ENTRY POINT = DSN8BCU2                                       *
                *       PURPOSE = SEE FUNCTION                                    *
                *       LINKAGE =                                                 *
                *          CALL 'DSN8BCU2' USING WORKAREA-IND RECWORK.           *
                *                                                                *
                *       INPUT   = SYMBOLIC LABEL/NAME = WORKAREA-IND             *
                *                 DESCRIPTION = INDICATOR VARIABLE ARRAY          *
                *                 01  WORKAREA-IND.                               *
                *                    02  WORKIND PIC S9(4) COMP OCCURS 750 TIMES. *
                *                                                                *
                *                 SYMBOLIC LABEL/NAME = RECWORK                   *
                *                 DESCRIPTION = WORK AREA FOR OUTPUT RECORD       *
                *                 01  RECWORK.                                    *
                *                    02  RECWORK-LEN PIC S9(8) COMP.              *
                *                                                                *
                *                 SYMBOLIC LABEL/NAME = SYSIN                     *
                *                 DESCRIPTION = INPUT REQUESTS - TABLE OR VIEW    *
                *                                                                *
```

Figure 243. Called program that does pointer manipulation (Part 1 of 10)

```
*        OUTPUT  = SYMBOLIC LABEL/NAME = SYSPRINT                *
*                  DESCRIPTION = PRINTED RESULTS                 *
*                                                                *
*                  SYMBOLIC LABEL/NAME = SYSREC01               *
*                  DESCRIPTION = UNLOADED TABLE DATA            *
*                                                                *
*    EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION              *
*    EXIT-ERROR =                                                *
*       RETURN CODE = NONE                                       *
*       ABEND CODES =  NONE                                      *
*       ERROR-MESSAGES =                                         *
*          DSNT490I SAMPLE COBOL DATA UNLOAD PROGRAM RELEASE 3.0*
*                     -  THIS IS THE HEADER, INDICATING A NORMAL *
*                     -  START FOR THIS PROGRAM.                 *
*          DSNT493I SQL ERROR, SQLCODE = NNNNNNNN               *
*                     -  AN SQL ERROR OR WARNING WAS ENCOUNTERED *
*                     -  ADDITIONAL INFORMATION FROM DSNTIAR     *
*                     -  FOLLOWS THIS MESSAGE.                   *
*          DSNT495I SUCCESSFUL UNLOAD XXXXXXXX ROWS OF          *
*                   TABLE TTTTTTTT                              *
*                     -  THE UNLOAD WAS SUCCESSFUL.  XXXXXXXX IS *
*                     -  THE NUMBER OF ROWS UNLOADED.  TTTTTTTT  *
*                     -  IS THE NAME OF THE TABLE OR VIEW FROM   *
*                     -  WHICH IT WAS UNLOADED.                  *
*          DSNT496I UNRECOGNIZED DATA TYPE CODE OF NNNNN        *
*                     -  THE PREPARE RETURNED AN INVALID DATA    *
*                     -  TYPE CODE.  NNNNN IS THE CODE, PRINTED  *
*                     -  IN DECIMAL.  USUALLY AN ERROR IN        *
*                     -  THIS ROUTINE OR A NEW DATA TYPE.        *
*          DSNT497I RETURN CODE FROM MESSAGE ROUTINE DSNTIAR    *
*                     -  THE MESSAGE FORMATTING ROUTINE DETECTED *
*                     -  AN ERROR.  SEE THAT ROUTINE FOR RETURN  *
*                     -  CODE INFORMATION.  USUALLY AN ERROR IN  *
*                     -  THIS ROUTINE.                           *
*          DSNT498I ERROR, NO VALID COLUMNS FOUND              *
*                     -  THE PREPARE RETURNED DATA WHICH DID NOT *
*                     -  PRODUCE A VALID OUTPUT RECORD.          *
*                     -  USUALLY AN ERROR IN THIS ROUTINE.       *
*          DSNT499I NO ROWS FOUND IN TABLE OR VIEW             *
*                     -  THE CHOSEN TABLE OR VIEWS DID NOT       *
*                     -  RETURN ANY ROWS.                        *
*          ERROR MESSAGES FROM MODULE DSNTIAR                   *
*                     -  WHEN AN ERROR OCCURS, THIS MODULE       *
*                     -  PRODUCES CORRESPONDING MESSAGES.        *
*                                                                *
*    EXTERNAL REFERENCES =                                       *
*       ROUTINES/SERVICES =                                      *
*          DSNTIAR  - TRANSLATE SQLCA INTO MESSAGES             *
*       DATA-AREAS    =     NONE                                 *
*       CONTROL-BLOCKS   =                                       *
*          SQLCA    - SQL COMMUNICATION AREA                    *
*                                                                *
*    TABLES = NONE                                               *
*    CHANGE-ACTIVITY = NONE                                      *
*                                                                *
```

Figure 243. Called program that does pointer manipulation (Part 2 of 10)

```
*  *PSEUDOCODE*                                                  *
*    PROCEDURE                                                   *
*    EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.               *
*    EXEC SQL DECLARE SEL STATEMENT END-EXEC.                   *
*    INITIALIZE THE DATA, OPEN FILES.                           *
*    OBTAIN STORAGE FOR THE SQLDA AND THE DATA RECORDS.         *
*    READ A TABLE NAME.                                         *
*    OPEN SYSREC01.                                             *
*    BUILD THE SQL STATEMENT TO BE EXECUTED                     *
*    EXEC SQL PREPARE SQL STATEMENT INTO SQLDA END-EXEC.        *
*    SET UP ADDRESSES IN THE SQLDA FOR DATA.                    *
*    INITIALIZE DATA RECORD COUNTER TO 0.                       *
*    EXEC SQL OPEN DT END-EXEC.                                 *
*    DO WHILE SQLCODE IS 0.                                     *
*    EXEC SQL FETCH DT USING DESCRIPTOR SQLDA END-EXEC.         *
*    ADD IN MARKERS TO DENOTE NULLS.                            *
*    WRITE THE DATA TO SYSREC01.                                *
*    INCREMENT DATA RECORD COUNTER.                             *
*    END.                                                        *
*    EXEC SQL CLOSE DT END-EXEC.                                *
*    INDICATE THE RESULTS OF THE UNLOAD OPERATION.             *
*    CLOSE THE SYSIN, SYSPRINT, AND SYSREC01 FILES.            *
*    END.                                                        *
*----------------------------------------------------------------*
/
 IDENTIFICATION DIVISION.
*-----------------------
 PROGRAM-ID.    DSN8BCU2
*
 ENVIRONMENT DIVISION.
*--------------------
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT SYSIN
            ASSIGN TO DA-S-SYSIN.
     SELECT SYSPRINT
            ASSIGN TO UT-S-SYSPRINT.
     SELECT SYSREC01
            ASSIGN TO DA-S-SYSREC01.
*
 DATA DIVISION.
*-------------
*
 FILE SECTION.
 FD     SYSIN
        RECORD CONTAINS 80 CHARACTERS
        BLOCK CONTAINS 0 RECORDS
        LABEL RECORDS ARE OMITTED
        RECORDING MODE IS F.
 01  CARDREC               PIC X(80).
*
 FD  SYSPRINT
        RECORD CONTAINS 120 CHARACTERS
        LABEL RECORDS ARE OMITTED
        DATA RECORD IS MSGREC
        RECORDING MODE IS F.
 01  MSGREC                PIC X(120).
```

Figure 243. Called program that does pointer manipulation (Part 3 of 10)

```
*
 FD  SYSREC01
         RECORD CONTAINS 5 TO 32704 CHARACTERS
         LABEL RECORDS ARE OMITTED
         DATA RECORD IS REC01
         RECORDING MODE IS V.
 01  REC01.
         02  REC01-LEN PIC S9(8) COMP.
         02  REC01-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
                          DEPENDING ON REC01-LEN.
/
 WORKING-STORAGE SECTION.
*
 ******************************************************
 * STRUCTURE FOR INPUT                                *
 ******************************************************
 01  IOAREA.
         02  TNAME         PIC X(72).
         02  FILLER        PIC X(08).
 01  STMTBUF.
         49  STMTLEN       PIC S9(4) COMP VALUE 92.
         49  STMTCHAR      PIC X(92).
 01  STMTBLD.
         02  FILLER        PIC X(20) VALUE 'SELECT * FROM'.
         02  STMTTAB       PIC X(72).
*
 ******************************************************
 * REPORT HEADER STRUCTURE                            *
 ******************************************************
 01  HEADER.
         02  FILLER PIC X(35)
             VALUE ' DSNT490I SAMPLE COBOL DATA UNLOAD '.
         02  FILLER PIC X(85) VALUE 'PROGRAM RELEASE 3.0'.
 01  MSG-SQLERR.
         02  FILLER PIC X(31)
             VALUE ' DSNT493I SQL ERROR, SQLCODE = '.
         02  MSG-MINUS      PIC X(1).
         02  MSG-PRINT-CODE PIC 9(8).
         02  FILLER PIC X(81) VALUE '                 '.
 01  UNLOADED.
         02  FILLER PIC X(28)
             VALUE ' DSNT495I SUCCESSFUL UNLOAD '.
         02  ROWS   PIC 9(8).
         02  FILLER PIC X(15) VALUE ' ROWS OF TABLE '.
         02  TABLENAM PIC X(72) VALUE '          '.
 01  BADTYPE.
         02  FILLER PIC X(42)
             VALUE ' DSNT496I UNRECOGNIZED DATA TYPE CODE OF  '.
         02  TYPCOD PIC 9(8).
         02  FILLER PIC X(71) VALUE '           '.
 01  MSGRETCD.
         02  FILLER PIC X(42)
             VALUE ' DSNT497I RETURN CODE FROM MESSAGE ROUTINE'.
         02  FILLER PIC X(9) VALUE 'DSNTIAR '.
         02  RETCODE    PIC 9(8).
         02  FILLER PIC X(62) VALUE '              '.
```

Figure 243. Called program that does pointer manipulation (Part 4 of 10)

```
 01  MSGNOCOL.
         02  FILLER PIC X(120)
             VALUE ' DSNT498I ERROR, NO VALID COLUMNS FOUND'.
 01  MSG-NOROW.
         02  FILLER PIC X(120)
             VALUE ' DSNT499I NO ROWS FOUND IN TABLE OR VIEW'.
*****************************************************
* WORKAREAS                                        *
*****************************************************
 77  NOT-FOUND        PIC S9(8) COMP VALUE  +100.
*****************************************************
* VARIABLES FOR ERROR-MESSAGE FORMATTING           *
00
*****************************************************
 01  ERROR-MESSAGE.
         02  ERROR-LEN   PIC S9(4)  COMP VALUE +960.
         02  ERROR-TEXT  PIC X(120) OCCURS 8 TIMES
                                    INDEXED BY ERROR-INDEX.
 77  ERROR-TEXT-LEN     PIC S9(8)  COMP VALUE +120.
*****************************************************
* SQL DESCRIPTOR AREA                              *
*****************************************************
         EXEC SQL INCLUDE SQLDA END-EXEC.
*
*  DATA TYPES FOUND IN SQLTYPE, AFTER REMOVING THE NULL BIT
*
 77  VARCTYPE          PIC S9(4)  COMP VALUE +448.
 77  CHARTYPE          PIC S9(4)  COMP VALUE +452.
 77  VARLTYPE          PIC S9(4)  COMP VALUE +456.
 77  VARGTYPE          PIC S9(4)  COMP VALUE +464.
 77  GTYPE             PIC S9(4)  COMP VALUE +468.
 77  LVARGTYP          PIC S9(4)  COMP VALUE +472.
 77  FLOATYPE          PIC S9(4)  COMP VALUE +480.
 77  DECTYPE           PIC S9(4)  COMP VALUE +484.
 77  INTTYPE           PIC S9(4)  COMP VALUE +496.
 77  HWTYPE            PIC S9(4)  COMP VALUE +500.
 77  DATETYP           PIC S9(4)  COMP VALUE +384.
 77  TIMETYP           PIC S9(4)  COMP VALUE +388.
 77  TIMESTMP          PIC S9(4)  COMP VALUE +392.
*
```

Figure 243. Called program that does pointer manipulation (Part 5 of 10)

```
     ******************************************************
     *   THE REDEFINES CLAUSES BELOW ARE FOR 31-BIT ADDRESSING.
     *   IF YOUR COMPILER SUPPORTS ONLY 24-BIT ADDRESSING,
     *   CHANGE THE DECLARATIONS TO THESE:
     *   01  RECNUM REDEFINES RECPTR PICTURE S9(8) COMPUTATIONAL.
     *   01 IRECNUM REDEFINES IRECPTR PICTURE S9(8) COMPUTATIONAL.
     ******************************************************
      01  RECPTR POINTER.
      01  RECNUM REDEFINES RECPTR PICTURE S9(9) COMPUTATIONAL.
      01  IRECPTR POINTER.
      01  IRECNUM REDEFINES IRECPTR PICTURE S9(9) COMPUTATIONAL.
      01  I      PICTURE S9(4) COMPUTATIONAL.
      01  J      PICTURE S9(4) COMPUTATIONAL.
      01  DUMMY  PICTURE S9(4) COMPUTATIONAL.
      01  MYTYPE PICTURE S9(4) COMPUTATIONAL.
      01  COLUMN-IND PICTURE S9(4) COMPUTATIONAL.
      01  COLUMN-LEN PICTURE S9(4) COMPUTATIONAL.
      01  COLUMN-PREC PICTURE S9(4) COMPUTATIONAL.
      01  COLUMN-SCALE PICTURE S9(4) COMPUTATIONAL.
      01  INDCOUNT           PIC S9(4) COMPUTATIONAL.
      01  ROWCOUNT           PIC S9(4) COMPUTATIONAL.
      01  WORKAREA2.
            02  WORKINDPTR  POINTER    OCCURS 750 TIMES.
     ******************************************************
     *   DECLARE CURSOR AND STATEMENT FOR DYNAMIC SQL
     ******************************************************
     *
             EXEC SQL DECLARE DT CURSOR FOR SEL  END-EXEC.
             EXEC SQL DECLARE SEL STATEMENT      END-EXEC.
     *
     ******************************************************
     * SQL INCLUDE FOR SQLCA                               *
     ******************************************************
             EXEC SQL INCLUDE SQLCA  END-EXEC.
     *
      77  ONE               PIC S9(4)  COMP VALUE +1.
      77  TWO               PIC S9(4)  COMP VALUE +2.
      77  FOUR              PIC S9(4)  COMP VALUE +4.
      77  QMARK             PIC X(1)        VALUE '?'.
     *
      LINKAGE SECTION.
      01  LINKAREA-IND.
            02  IND   PIC   S9(4) COMP OCCURS 750 TIMES.
      01  LINKAREA-REC.
            02  REC1-LEN PIC S9(8) COMP.
            02  REC1-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
                           DEPENDING ON REC1-LEN.
      01  LINKAREA-QMARK.
            02  INDREC PIC   X(1).
     /
```

Figure 243. Called program that does pointer manipulation (Part 6 of 10)

```
      PROCEDURE DIVISION USING LINKAREA-IND LINKAREA-REC.
*
*****************************************************
* SQL RETURN CODE HANDLING                         *
*****************************************************
      EXEC SQL WHENEVER SQLERROR   GOTO DBERROR END-EXEC.
      EXEC SQL WHENEVER SQLWARNING GOTO DBERROR END-EXEC.
      EXEC SQL WHENEVER NOT FOUND  CONTINUE     END-EXEC.
*
*****************************************************
* MAIN PROGRAM ROUTINE                             *
*****************************************************
          SET IRECPTR TO ADDRESS OF REC1-CHAR(1).
*                                          **OPEN FILES
          OPEN INPUT  SYSIN
               OUTPUT SYSPRINT
               OUTPUT SYSREC01.
*                                          **WRITE HEADER
          WRITE MSGREC FROM HEADER
               AFTER ADVANCING 2 LINES.
*                                          **GET FIRST INPUT
          READ SYSIN  RECORD INTO IOAREA.
*                                          **MAIN ROUTINE
          PERFORM PROCESS-INPUT THROUGH IND-RESULT.
*
 PROG-END.
*                                          **CLOSE FILES
          CLOSE SYSIN
                SYSPRINT
                SYSREC01.
          GOBACK.
/
***************************************************************
*                                                           *
*    PERFORMED SECTION:                                     *
*    PROCESSING FOR THE TABLE OR VIEW JUST READ            *
*                                                           *
***************************************************************
 PROCESS-INPUT.
*
     MOVE TNAME TO STMTTAB.
     MOVE STMTBLD TO STMTCHAR.
     EXEC SQL PREPARE SEL INTO :SQLDA FROM :STMTBUF  END-EXEC.
***************************************************************
*                                                           *
*    SET UP ADDRESSES IN THE SQLDA FOR DATA.               *
*                                                           *
***************************************************************
     IF SQLD = ZERO THEN
        WRITE MSGREC FROM MSGNOCOL
              AFTER ADVANCING 2 LINES
        GO TO IND-RESULT.
     MOVE ZERO TO ROWCOUNT.
     MOVE ZERO TO REC1-LEN.
     SET RECPTR TO IRECPTR.
     MOVE ONE TO I.
     PERFORM COLADDR UNTIL I > SQLD.
```

Figure 243. Called program that does pointer manipulation (Part 7 of 10)

```
          ****************************************************************
          *                                                              *
          *    SET LENGTH OF OUTPUT RECORD.                              *
          *    EXEC  SQL OPEN DT END-EXEC.                               *
          *    DO WHILE SQLCODE IS 0.                                    *
          *       EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.  *
          *       ADD IN MARKERS TO DENOTE NULLS.                       *
          *       WRITE THE DATA TO SYSREC01.                           *
          *       INCREMENT DATA RECORD COUNTER.                        *
          *    END.                                                      *
          *                                                              *
          ****************************************************************
          *                                          **OPEN CURSOR
               EXEC SQL OPEN DT  END-EXEC.
               PERFORM BLANK-REC.
               EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
          *                                          **NO ROWS FOUND
          *                                          **PRINT ERROR MESSAGE
                   IF SQLCODE = NOT-FOUND
                      WRITE MSGREC FROM MSG-NOROW
                         AFTER ADVANCING 2 LINES
                   ELSE
          *                                          **WRITE ROW AND
          *                                          **CONTINUE UNTIL
          *                                          **NO MORE ROWS
                      PERFORM WRITE-AND-FETCH
                         UNTIL SQLCODE IS NOT EQUAL TO ZERO.
          *
               EXEC SQL WHENEVER NOT FOUND  GOTO CLOSEDT    END-EXEC.
          *
           CLOSEDT.
               EXEC SQL CLOSE DT  END-EXEC.
          *
          ****************************************************************
          *                                                              *
          *    INDICATE THE RESULTS OF THE UNLOAD OPERATION.            *
          *                                                              *
          ****************************************************************
           IND-RESULT.
                   MOVE TNAME TO TABLENAM.
                   MOVE ROWCOUNT TO ROWS.
                   WRITE MSGREC FROM UNLOADED
                        AFTER ADVANCING 2 LINES.
                   GO TO PROG-END.
          *
           WRITE-AND-FETCH.
          *       ADD IN MARKERS TO DENOTE NULLS.
                   MOVE ONE TO INDCOUNT.
                   PERFORM NULLCHK UNTIL INDCOUNT = SQLD.
                   MOVE REC1-LEN TO REC01-LEN.
                   WRITE REC01 FROM LINKAREA-REC.
                   ADD ONE TO ROWCOUNT.
                   PERFORM BLANK-REC.
                   EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
          *
           NULLCHK.
                   IF IND(INDCOUNT) < 0 THEN
                       SET ADDRESS OF LINKAREA-QMARK TO WORKINDPTR(INDCOUNT)
                       MOVE QMARK TO INDREC.
                   ADD ONE TO INDCOUNT.
```

Figure 243. Called program that does pointer manipulation (Part 8 of 10)

```
*****************************************************
*    BLANK OUT RECORD TEXT FIRST                    *
*****************************************************
 BLANK-REC.
         MOVE ONE TO J.
         PERFORM BLANK-MORE UNTIL J > REC1-LEN.
 BLANK-MORE.
         MOVE ' ' TO REC1-CHAR(J).
         ADD  ONE TO J.
*
 COLADDR.
     SET SQLDATA(I) TO RECPTR.
*****************************************************************
*
*       DETERMINE THE LENGTH OF THIS COLUMN (COLUMN-LEN)
*       THIS DEPENDS UPON THE DATA TYPE.  MOST DATA TYPES HAVE
*       THE LENGTH SET, BUT VARCHAR, GRAPHIC, VARGRAPHIC, AND
*       DECIMAL DATA NEED TO HAVE THE BYTES CALCULATED.
*       THE NULL ATTRIBUTE MUST BE SEPARATED TO SIMPLIFY MATTERS.
*
*****************************************************************
     MOVE SQLLEN(I) TO COLUMN-LEN.
*       COLUMN-IND IS 0 FOR NO NULLS AND 1 FOR NULLS
     DIVIDE SQLTYPE(I) BY TWO GIVING DUMMY REMAINDER COLUMN-IND.
*       MYTYPE IS JUST THE SQLTYPE WITHOUT THE NULL BIT
     MOVE SQLTYPE(I) TO MYTYPE.
     SUBTRACT COLUMN-IND FROM MYTYPE.
*       SET THE COLUMN LENGTH, DEPENDENT UPON DATA TYPE
     EVALUATE MYTYPE
         WHEN     CHARTYPE  CONTINUE,
         WHEN     DATETYP   CONTINUE,
         WHEN     TIMETYP   CONTINUE,
         WHEN     TIMESTMP  CONTINUE,
         WHEN     FLOATYPE  CONTINUE,
         WHEN     VARCTYPE
             ADD TWO TO COLUMN-LEN,
         WHEN     VARLTYPE
             ADD TWO TO COLUMN-LEN,
         WHEN     GTYPE
             MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN,
         WHEN     VARGTYPE
             PERFORM CALC-VARG-LEN,
         WHEN     LVARGTYP
             PERFORM CALC-VARG-LEN,
         WHEN     HWTYPE
             MOVE TWO TO COLUMN-LEN,
         WHEN     INTTYPE
             MOVE FOUR TO COLUMN-LEN,
         WHEN     DECTYPE
             PERFORM CALC-DECIMAL-LEN,
         WHEN     OTHER
             PERFORM UNRECOGNIZED-ERROR,
     END-EVALUATE.
     ADD COLUMN-LEN TO RECNUM.
     ADD COLUMN-LEN TO REC1-LEN.
```

Figure 243. Called program that does pointer manipulation (Part 9 of 10)

```
     ****************************************************************
     *                                                              *
     *    IF THIS COLUMN CAN BE NULL, AN INDICATOR VARIABLE IS      *
     *    NEEDED.  WE ALSO RESERVE SPACE IN THE OUTPUT RECORD TO    *
     *    NOTE THAT THE VALUE IS NULL.                              *
     *                                                              *
     ****************************************************************
          MOVE ZERO TO IND(I).
          IF COLUMN-IND = ONE THEN
             SET SQLIND(I) TO ADDRESS OF IND(I)
             SET WORKINDPTR(I) TO RECPTR
             ADD ONE TO RECNUM
             ADD ONE TO REC1-LEN.
     *
          ADD ONE TO I.
     *      PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
     *      FOR A DECIMAL DATA TYPE COLUMN
      CALC-DECIMAL-LEN.
             DIVIDE COLUMN-LEN BY 256 GIVING COLUMN-PREC
                     REMAINDER COLUMN-SCALE.
             MOVE COLUMN-PREC TO COLUMN-LEN.
             ADD ONE TO COLUMN-LEN.
             DIVIDE COLUMN-LEN BY TWO GIVING COLUMN-LEN.
     *      PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
     *      FOR A VARGRAPHIC DATA TYPE COLUMN
      CALC-VARG-LEN.
             MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN.
             ADD TWO TO COLUMN-LEN.
     *      PERFORMED PARAGRAPH TO NOTE AN UNRECOGNIZED
     *      DATA TYPE COLUMN
      UNRECOGNIZED-ERROR.
     *
     *      ERROR MESSAGE FOR UNRECOGNIZED DATA TYPE
     *
             MOVE  SQLTYPE(I) TO TYPCOD.
                WRITE MSGREC FROM BADTYPE
                    AFTER ADVANCING 2 LINES.
             GO TO IND-RESULT.
     *
     *****************************************************
     * SQL ERROR OCCURRED - GET MESSAGE                  *
     *****************************************************
      DBERROR.
     *                                    **SQL ERROR
             MOVE SQLCODE TO MSG-PRINT-CODE.
             IF SQLCODE < 0 THEN MOVE '-' TO MSG-MINUS.
             WRITE MSGREC FROM MSG-SQLERR
                AFTER ADVANCING 2 LINES.
             CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
             IF RETURN-CODE = ZERO
                PERFORM ERROR-PRINT VARYING ERROR-INDEX
                    FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8
             ELSE
     *                                    **ERROR FOUND IN DSNTIAR
     *                                    **PRINT ERROR MESSAGE
             MOVE RETURN-CODE TO RETCODE
             WRITE MSGREC FROM MSGRETCD
                AFTER ADVANCING 2 LINES.
             GO TO PROG-END.
     *
     *****************************************************
     *    PRINT MESSAGE TEXT                             *
     *****************************************************
      ERROR-PRINT.
             WRITE MSGREC FROM ERROR-TEXT (ERROR-INDEX)
                 AFTER ADVANCING 1 LINE.
```

Figure 243. Called program that does pointer manipulation (Part 10 of 10)

Sample dynamic and static SQL in a C program

Figure 244 illustrates dynamic SQL and static SQL embedded in a C program. Each section of the program is identified with a comment. Section 1 of the program shows static SQL; sections 2, 3, and 4 show dynamic SQL. The function of each section is explained in detail in the prologue to the program.

```
/*********************************************************************/
/*  Descriptive name = Dynamic SQL sample using C language          */
/*                                                                   */
/*  Function = To show examples of the use of dynamic and static     */
/*             SQL.                                                   */
/*                                                                   */
/*  Notes = This example assumes that the EMP and DEPT tables are    */
/*          defined. They need not be the same as the DB2 Sample     */
/*          tables.                                                   */
/*                                                                   */
/*  Module type    = C program                                       */
/*     Processor    = DB2  precompiler, C compiler                   */
/*     Module size = see link edit                                   */
/*     Attributes  = not reentrant or reusable                       */
/*                                                                   */
/*     Input    =                                                    */
/*                                                                   */
/*              symbolic label/name = DEPT                           */
/*              description = arbitrary table                        */
/*              symbolic label/name = EMP                            */
/*              description = arbitrary table                        */
/*                                                                   */
/*     Output   =                                                    */
/*                                                                   */
/*              symbolic label/name = SYSPRINT                       */
/*              description = print results via printf               */
/*                                                                   */
/*  Exit-normal = return code 0 normal completion                   */
/*                                                                   */
/*  Exit-error =                                                     */
/*                                                                   */
/*     Return code    = SQLCA                                        */
/*                                                                   */
/*     Abend codes    =  none                                        */
/*                                                                   */
/*  External references  = none                                      */
/*                                                                   */
/*     Control-blocks    =                                           */
/*          SQLCA    - sql communication area                        */
/*                                                                   */
```

Figure 244. Sample SQL in a C program (Part 1 of 4)

```
/* Logic specification:                                                  */
/*                                                                        */
/* There are four SQL sections.                                          */
/*                                                                        */
/* 1) STATIC SQL  1: using static cursor with a SELECT statement.        */
/*    Two output host variables.                                         */
/* 2) Dynamic SQL 2: Fixed-list SELECT, using same SELECT statement      */
/*    used in SQL 1 to show the difference. The prepared string          */
/*    :iptstr can be assigned with other dynamic-able SQL statements. */
/* 3) Dynamic SQL 3: Insert with parameter markers.                      */
/*    Using four parameter markers which represent four input host       */
/*    variables within a host structure.                                 */
/* 4) Dynamic SQL 4: EXECUTE IMMEDIATE                                   */
/*    A GRANT statement is executed immediately by passing it to DB2   */
/*    via a varying string host variable. The example shows how to       */
/*    set up the host variable before passing it.                        */
/*                                                                        */
/************************************************************************/

 #include "stdio.h"
 #include "stdefs.h"
 EXEC SQL INCLUDE SQLCA;
 EXEC SQL INCLUDE SQLDA;
 EXEC SQL BEGIN DECLARE SECTION;
 short edlevel;
 struct { short len;
          char  x1[56];
        } stmtbf1, stmtbf2, inpstr;
 struct { short len;
          char  x1[15];
        } lname;
 short hv1;
 struct { char  deptno[4];
          struct { short len;
                   char  x[36];
                 } deptname;
          char  mgrno[7];
          char  admrdept[4];
        } hv2;
 short ind[4];
 EXEC SQL END   DECLARE SECTION;
 EXEC SQL DECLARE EMP TABLE
             (EMPNO          CHAR(6)               ,
              FIRSTNAME      VARCHAR(12)           ,
              MIDINIT        CHAR(1)               ,
              LASTNAME       VARCHAR(15)           ,
              WORKDEPT       CHAR(3)               ,
              PHONENO        CHAR(4)               ,
              HIREDATE       DECIMAL(6)            ,
              JOBCODE        DECIMAL(3)            ,
              EDLEVEL        SMALLINT              ,
              SEX            CHAR(1)               ,
              BIRTHDATE      DECIMAL(6)            ,
              SALARY         DECIMAL(8,2)          ,
              FORFNAME       VARGRAPHIC(12)        ,
              FORMNAME       GRAPHIC(1)            ,
              FORLNAME       VARGRAPHIC(15)        ,
              FORADDR        VARGRAPHIC(256) )     ;
```

Figure 244. Sample SQL in a C program (Part 2 of 4)

```
            EXEC SQL DECLARE DEPT TABLE
                        (
                        DEPTNO          CHAR(3)               ,
                        DEPTNAME        VARCHAR(36)           ,
                        MGRNO           CHAR(6)               ,
                        ADMRDEPT        CHAR(3)               );
main ()
{
printf("??/n***      begin of program                        ***");
EXEC SQL WHENEVER SQLERROR GO TO HANDLERR;
EXEC SQL WHENEVER SQLWARNING GO TO HANDWARN;
EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND;
/*****************************************************************/
/* Assign values to host variables which will be input to DB2    */
/*****************************************************************/
strcpy(hv2.deptno,"M92");
strcpy(hv2.deptname.x,"DDL");
hv2.deptname.len = strlen(hv2.deptname.x);
strcpy(hv2.mgrno,"123456");
strcpy(hv2.admrdept,"abc");
/*****************************************************************/
/* Static  SQL 1:  DECLARE CURSOR, OPEN, FETCH, CLOSE            */
/* Select into :edlevel, :lname                                 */
/*****************************************************************/
printf("??/n***      begin declare                           ***");
EXEC SQL DECLARE C1 CURSOR FOR SELECT EDLEVEL, LASTNAME FROM EMP
        WHERE EMPNO = '000010';
printf("??/n***      begin open                              ***");
EXEC SQL OPEN C1;

printf("??/n***      begin fetch                             ***");
EXEC SQL FETCH C1 INTO :edlevel, :lname;
printf("??/n*** returned values                              ***");
printf("??/n??/nedlevel = %d",edlevel);
printf("??/nlname = %s\n",lname.x1);

printf("??/n***      begin close                             ***");
EXEC SQL CLOSE C1;
/*****************************************************************/
/* Dynamic SQL 2:  PREPARE, DECLARE CURSOR, OPEN, FETCH, CLOSE    */
/* Select into :edlevel, :lname                                 */
/*****************************************************************/
sprintf (inpstr.x1,
        "SELECT EDLEVEL, LASTNAME FROM EMP WHERE EMPNO = '000010'");
inpstr.len = strlen(inpstr.x1);
printf("??/n***      begin prepare                           ***");
EXEC SQL PREPARE STAT1 FROM :inpstr;
printf("??/n***      begin declare                           ***");
EXEC SQL DECLARE C2 CURSOR FOR STAT1;
printf("??/n***      begin open                              ***");
EXEC SQL OPEN C2;

printf("??/n***      begin fetch                             ***");
EXEC SQL FETCH C2 INTO :edlevel, :lname;
printf("??/n*** returned values                              ***");
printf("??/n??/nedlevel = %d",edlevel);
printf("??/nlname = %s\n",lname.x1);

printf("??/n***      begin close                             ***");
EXEC SQL CLOSE C2;
```

Figure 244. Sample SQL in a C program (Part 3 of 4)

```
/******************************************************************/
/* Dynamic SQL 3:  PREPARE with parameter markers                 */
/* Insert into with four values.                                  */
/******************************************************************/
sprintf (stmtbf1.x1,
         "INSERT INTO DEPT VALUES (?,?,?,?)");
stmtbf1.len = strlen(stmtbf1.x1);
printf("??/n***      begin prepare                    ***");
EXEC SQL PREPARE s1 FROM :stmtbf1;
printf("??/n***      begin execute                    ***");
EXEC SQL EXECUTE s1 USING :hv2:ind;
printf("??/n***   following are expected insert results    ***");
printf("??/n  hv2.deptno = %s",hv2.deptno);
printf("??/n  hv2.deptname.len = %d",hv2.deptname.len);
printf("??/n  hv2.deptname.x = %s",hv2.deptname.x);
printf("??/n  hv2.mgrno = %s",hv2.mgrno);
printf("??/n  hv2.admrdept = %s",hv2.admrdept);
EXEC SQL COMMIT;
/******************************************************************/
/* Dynamic SQL 4:  EXECUTE IMMEDIATE                              */
/* Grant select                                                   */
/******************************************************************/
sprintf (stmtbf2.x1,
         "GRANT SELECT ON EMP TO USERX");
stmtbf2.len = strlen(stmtbf2.x1);
printf("??/n***      begin execute immediate          ***");
EXEC SQL EXECUTE IMMEDIATE :stmtbf2;
printf("??/n***      end of program                   ***");
goto progend;
HANDWARN: HANDLERR: NOTFOUND: ;
printf("??/n  SQLCODE  = %d",SQLCODE);
printf("??/n  SQLWARN0 = %c",SQLWARN0);
printf("??/n  SQLWARN1 = %c",SQLWARN1);
printf("??/n  SQLWARN2 = %c",SQLWARN2);
printf("??/n  SQLWARN3 = %c",SQLWARN3);
printf("??/n  SQLWARN4 = %c",SQLWARN4);
printf("??/n  SQLWARN5 = %c",SQLWARN5);
printf("??/n  SQLWARN6 = %c",SQLWARN6);
printf("??/n  SQLWARN7 = %c",SQLWARN7);
progend: ;
}
```

Figure 244. Sample SQL in a C program (Part 4 of 4)

Example DB2 REXX application

The following example shows a complete DB2 REXX application named DRAW. DRAW must be invoked from the command line of an ISPF edit session. DRAW takes a table or view name as input and produces a SELECT, INSERT, or UPDATE SQL statement or a LOAD utility control statement that includes the columns of the table as output.

DRAW syntax:

```
►►──%DRAW──object-name──(─┬───────────────┬─┬──────────────────────────┬──────────────────►◄
                         └─SSID=ssid──┘   │        ┌─SELECT─┐            │
                                          └─TYPE=──┼─INSERT─┼────────────┘
                                                   ├─UPDATE─┤
                                                   └─LOAD───┘
```
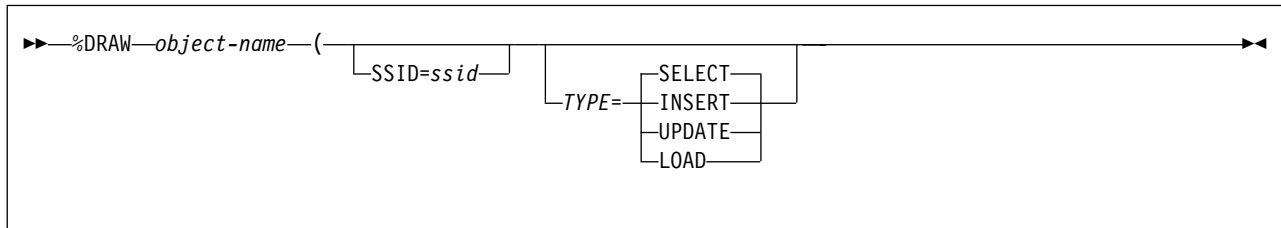
DRAW parameters:

object-name
> The name of the table or view for which DRAW builds an SQL statement or utility control statement. The name can be a one-, two-, or three-part name. The table or view to which *object-name* refers must exist before DRAW can run.
>
> *object-name* is a required parameter.

SSID=*ssid*
> Specifies the name of the local DB2 subsystem.
>
> S can be used as an abbreviation for SSID.
>
> If you invoke DRAW from the command line of the edit session in SPUFI, SSID=*ssid* is an optional parameter. DRAW uses the subsystem ID from the DB2I Defaults panel.

TYPE=*operation-type*
> The type of statement that DRAW builds.
>
> T can be used as an abbreviation for TYPE.
>
> *operation-type* has one of the following values:

> | | |
> |---|---|
> | **SELECT** | Builds a SELECT statement in which the result table contains all columns of *object-name*. |
> | | S can be used as an abbreviation for SELECT. |
> | **INSERT** | Builds a template for an INSERT statement that inserts values into all columns of *object-name*. The template contains comments that indicate where the user can place column values. |
> | | I can be used as an abbreviation for INSERT. |
> | **UPDATE** | Builds a template for an UPDATE statement that updates columns of *object-name*. The template contains comments that indicate where the user can place column values and qualify the update operation for selected rows. |
> | | U can be used as an abbreviation for UPDATE. |
> | **LOAD** | Builds a template for a LOAD utility control statement for *object-name*. |
> | | L can be used as an abbreviation for LOAD. |

> TYPE=*operation-type* is an optional parameter. The default is TYPE=SELECT.

DRAW data sets:

Edit data set

The data set from which you issue the DRAW command when you are in an ISPF edit session. If you issue the DRAW command from a SPUFI session, this data set is the data set that you specify in field 1 of the main SPUFI panel (DSNESP01). The output from the DRAW command goes into this data set.

DRAW return codes:

Return code	Meaning
0	Successful completion.
12	An error occurred when DRAW edited the input file.
20	One of the following errors occurred: • No input parameters were specified. • One of the input parameters was not valid. • An SQL error occurred when the output statement was generated.

Examples of DRAW invocation:

Generate a SELECT statement for table DSN8710.EMP at the local subsystem. Use the default DB2I subsystem ID.

The DRAW invocation is:
```
DRAW DSN8710.EMP (TYPE=SELECT
```

The output is:
```
SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
  "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
  "SALARY" , "BONUS" , "COMM"
FROM DSN8710.EMP
```

Generate a template for an INSERT statement that inserts values into table DSN8710.EMP at location SAN_JOSE. The local subsystem ID is DSN.

The DRAW invocation is:
```
DRAW SAN_JOSE.DSN8710.EMP (TYPE=INSERT SSID=DSN
```

The output is:
```
INSERT INTO SAN_JOSE.DSN8710.EMP  ( "EMPNO" , "FIRSTNME" , "MIDINIT" ,
  "LASTNAME" ,  "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" ,
  "EDLEVEL" , "SEX" , "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
  VALUES (
-- ENTER VALUES BELOW          COLUMN NAME          DATA TYPE
                        , -- EMPNO                CHAR(6) NOT NULL
                        , -- FIRSTNME             VARCHAR(12) NOT NULL
                        , -- MIDINIT              CHAR(1) NOT NULL
                        , -- LASTNAME             VARCHAR(15) NOT NULL
                        , -- WORKDEPT             CHAR(3)
                        , -- PHONENO              CHAR(4)
                        , -- HIREDATE             DATE
                        , -- JOB                  CHAR(8)
                        , -- EDLEVEL              SMALLINT
                        , -- SEX                  CHAR(1)
                        , -- BIRTHDATE            DATE
                        , -- SALARY               DECIMAL(9,2)
                        , -- BONUS                DECIMAL(9,2)
                        ) -- COMM                 DECIMAL(9,2)
```

Generate a template for an UPDATE statement that updates values of table
DSN8710.EMP. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW DSN8710.EMP (TYPE=UPDATE SSID=DSN
```

The output is:

```
UPDATE DSN8710.EMP  SET
-- COLUMN NAME            ENTER VALUES BELOW      DATA TYPE
   "EMPNO"=                                    -- CHAR(6) NOT NULL
 , "FIRSTNME"=                                 -- VARCHAR(12) NOT NULL
 , "MIDINIT"=                                  -- CHAR(1) NOT NULL
 , "LASTNAME"=                                 -- VARCHAR(15) NOT NULL
 , "WORKDEPT"=                                 -- CHAR(3)
 , "PHONENO"=                                  -- CHAR(4)
 , "HIREDATE"=                                 -- DATE
 , "JOB"=                                      -- CHAR(8)
 , "EDLEVEL"=                                  -- SMALLINT
 , "SEX"=                                      -- CHAR(1)
 , "BIRTHDATE"=                                -- DATE
 , "SALARY"=                                   -- DECIMAL(9,2)
 , "BONUS"=                                    -- DECIMAL(9,2)
 , "COMM"=                                     -- DECIMAL(9,2)
WHERE
```

Generate a LOAD control statement to load values into table DSN8710.EMP. The
local subsystem ID is DSN.

The draw invocation is:

```
DRAW DSN8710.EMP (TYPE=LOAD SSID=DSN
```

The output is:

```
LOAD DATA INDDN SYSREC INTO TABLE DSN8710.EMP
 ( "EMPNO"            POSITION(    1) CHAR(6)
 , "FIRSTNME"         POSITION(    8) VARCHAR
 , "MIDINIT"          POSITION(   21) CHAR(1)
 , "LASTNAME"         POSITION(   23) VARCHAR
 , "WORKDEPT"         POSITION(   39) CHAR(3)
                        NULLIF(   39)='?'
 , "PHONENO"          POSITION(   43) CHAR(4)
                        NULLIF(   43)='?'
 , "HIREDATE"         POSITION(   48) DATE EXTERNAL
                        NULLIF(   48)='?'
 , "JOB"              POSITION(   59) CHAR(8)
                        NULLIF(   59)='?'
 , "EDLEVEL"          POSITION(   68) SMALLINT
                        NULLIF(   68)='?'
 , "SEX"              POSITION(   71) CHAR(1)
                        NULLIF(   71)='?'
 , "BIRTHDATE"        POSITION(   73) DATE EXTERNAL
                        NULLIF(   73)='?'
 , "SALARY"           POSITION(   84) DECIMAL EXTERNAL(9,2)
                        NULLIF(   84)='?'
 , "BONUS"            POSITION(   90) DECIMAL EXTERNAL(9,2)
                        NULLIF(   90)='?'
 , "COMM"             POSITION(   96) DECIMAL EXTERNAL(9,2)
                        NULLIF(   96)='?'
 )
```

DRAW source code:

```
/* REXX ****************************************************************/
L1 = WHEREAMI()
/*
DRAW creates basic SQL queries by retrieving the description of a
table.  You must specify the name of the table or view to be queried.
You can specify the type of query you want to compose.  You might need
to specify the name of the DB2 subsystem.

>>--DRAW-----tablename-----|----------------------------|------->><
                           |-(-|-Ssid=subsystem-name-|-|
                           |       +-Select-+         |
                           |-Type=-|-Insert-|----|
                           |       |-Update-|
                           |       +--Load--+
```

Ssid=subsystem-name
    subsystem-name specified the name of a DB2 subsystem.

Select
    Composes a basic query for selecting data from the columns of a
    table or view. If TYPE is not specified, SELECT is assumed.
    Using SELECT with the DRAW command produces a query that would
    retrieve all rows and all columns from the specified table. You
    can then modify the query as needed.

    A SELECT query of EMP composed by DRAW looks like this:

```
SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
       "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
       "SALARY" , "BONUS" , "COMM"
FROM DSN8710.EMP
```

            If you include a location qualifier, the query looks like this:

```
SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
       "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
       "SALARY" , "BONUS" , "COMM"
FROM STLEC1.DSN8710.EMP
```

Figure 245. REXX sample program DRAW (Part 1 of 10)

To use this SELECT query, type the other clauses you need. If
you are selecting from more than one table, use a DRAW command
for each table name you want represented.

Insert
    Composes a basic query to insert data into the columns of a table
    or view.

    The following example shows an INSERT query of EMP that
    DRAW composed:

```
INSERT INTO DSN8710.EMP  ( "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" ,
      "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" ,
      "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
   VALUES (
-- ENTER VALUES BELOW          COLUMN NAME          DATA TYPE
                    , -- EMPNO                CHAR(6) NOT NULL
                    , -- FIRSTNME             VARCHAR(12) NOT NULL
                    , -- MIDINIT              CHAR(1) NOT NULL
                    , -- LASTNAME             VARCHAR(15) NOT NULL
                    , -- WORKDEPT             CHAR(3)
                    , -- PHONENO              CHAR(4)
                    , -- HIREDATE             DATE
                    , -- JOB                  CHAR(8)
                    , -- EDLEVEL              SMALLINT
                    , -- SEX                  CHAR(1)
                    , -- BIRTHDATE            DATE
                    , -- SALARY               DECIMAL(9,2)
                    , -- BONUS                DECIMAL(9,2)
                    ) -- COMM                 DECIMAL(9,2)
```

    To insert values into EMP, type values to the left of the
    column names. See *DB2 SQL Reference* for more information on
    INSERT queries.

Update
    Composes a basic query to change the data in a table or view.

    The following example shows an UPDATE query of EMP composed
    by DRAW:

Figure 245. REXX sample program DRAW (Part 2 of 10)

```
UPDATE DSN8710.EMP  SET
-- COLUMN NAME              ENTER VALUES BELOW       DATA TYPE
   "EMPNO"=                                       -- CHAR(6) NOT NULL
 , "FIRSTNME"=                                    -- VARCHAR(12) NOT NULL
 , "MIDINIT"=                                     -- CHAR(1) NOT NULL
 , "LASTNAME"=                                    -- VARCHAR(15) NOT NULL
 , "WORKDEPT"=                                    -- CHAR(3)
 , "PHONENO"=                                     -- CHAR(4)
 , "HIREDATE"=                                    -- DATE
 , "JOB"=                                         -- CHAR(8)
 , "EDLEVEL"=                                     -- SMALLINT
 , "SEX"=                                         -- CHAR(1)
 , "BIRTHDATE"=                                   -- DATE
 , "SALARY"=                                      -- DECIMAL(9,2)
 , "BONUS"=                                       -- DECIMAL(9,2)
 , "COMM"=                                        -- DECIMAL(9,2)
WHERE

    To use this UPDATE query, type the changes you want to make to
    the right of the column names, and delete the lines you don't
    need.  Be sure to complete the WHERE clause.  For information on
    writing queries to update data, refer to DB2 SQL Reference.

Load
    Composes a load statement to load the data in a table.

    The following example shows a LOAD statement of EMP composed
    by DRAW:

LOAD DATA INDDN SYSREC INTO TABLE DSN8710.EMP
 ( "EMPNO"              POSITION(    1) CHAR(6)
 , "FIRSTNME"           POSITION(    8) VARCHAR
 , "MIDINIT"            POSITION(   21) CHAR(1)
 , "LASTNAME"           POSITION(   23) VARCHAR
 , "WORKDEPT"           POSITION(   39) CHAR(3)
                        NULLIF(   39)='?'
 , "PHONENO"            POSITION(   43) CHAR(4)
                        NULLIF(   43)='?'
 , "HIREDATE"           POSITION(   48) DATE EXTERNAL
                        NULLIF(   48)='?'
 , "JOB"                POSITION(   59) CHAR(8)
                        NULLIF(   59)='?'
 , "EDLEVEL"            POSITION(   68) SMALLINT
                        NULLIF(   68)='?'
 , "SEX"                POSITION(   71) CHAR(1)
                        NULLIF(   71)='?'
 , "BIRTHDATE"          POSITION(   73) DATE EXTERNAL
                        NULLIF(   73)='?'
 , "SALARY"             POSITION(   84) DECIMAL EXTERNAL(9,2)
                        NULLIF(   84)='?'
 , "BONUS"              POSITION(   90) DECIMAL EXTERNAL(9,2)
                        NULLIF(   90)='?'
 , "COMM"               POSITION(   96) DECIMAL EXTERNAL(9,2)
                        NULLIF(   96)='?'
 )
```

Figure 245. REXX sample program DRAW (Part 3 of 10)

```
        To use this LOAD statement, type the changes you want to make,
        and delete the lines you don't need.  For information on writing
        queries to update data, refer to
        DB2 Utility Guide and Reference.

*/
   L2 = WHEREAMI()
/*******************************************************************/
/* TRACE ?R                                                        */
/*******************************************************************/
Address ISPEXEC
"ISREDIT MACRO (ARGS) NOPROCESS"
If ARGS = "" Then
Do
  Do I = L1+2 To L2-2;Say SourceLine(I);End
  Exit (20)
End
Parse Upper Var Args Table "(" Parms
Parms = Translate(Parms," ",",")
Type = "SELECT" /* Default */
SSID = ""        /* Default */
"VGET (DSNEOV01)"
If RC = 0 Then SSID = DSNEOV01
If (Parms <> "") Then
Do Until(Parms = "")
Parse Var Parms Var "=" Value Parms
  If Var = "T" | Var = "TYPE" Then Type = Value
  Else
  If Var = "S" | Var = "SSID" Then SSID = Value
  Else
    Exit (20)
End
"CONTROL ERRORS RETURN"
"ISREDIT (LEFTBND,RIGHTBND) = BOUNDS"
"ISREDIT (LRECL) = DATA_WIDTH" /*LRECL*/
BndSize = RightBnd - LeftBnd + 1
If BndSize > 72 Then BndSize = 72
"ISREDIT PROCESS DEST"
Select
  When rc = 0 Then
    'ISREDIT (ZDEST) = LINENUM .ZDEST'
  When rc <= 8 Then /* No A or B entered */
    Do
      zedsmsg = 'Enter "A"/"B" line cmd'
      zedlmsg = 'DRAW requires an "A" or "B" line command'
      'SETMSG MSG(ISRZ001)'
      Exit 12
    End
  When rc < 20 Then /* Conflicting line commands - edit sets message */
    Exit 12
  When rc = 20 Then
    zdest = 0
  Otherwise
    Exit 12
End
```

Figure 245. REXX sample program DRAW (Part 4 of 10)

```
SQLTYPE. = "UNKNOWN TYPE"
VCHTYPE  = 448; SQLTYPES.VCHTYPE  = 'VARCHAR'
CHTYPE   = 452; SQLTYPES.CHTYPE   = 'CHAR'
LVCHTYPE = 456; SQLTYPES.LVCHTYPE = 'VARCHAR'
VGRTYP   = 464; SQLTYPES.VGRTYP   = 'VARGRAPHIC'
GRTYP    = 468; SQLTYPES.GRTYP    = 'GRAPHIC'
LVGRTYP  = 472; SQLTYPES.LVGRTYP  = 'VARGRAPHIC'
FLOTYPE  = 480; SQLTYPES.FLOTYPE  = 'FLOAT'
DCTYPE   = 484; SQLTYPES.DCTYPE   = 'DECIMAL'
INTYPE   = 496; SQLTYPES.INTYPE   = 'INTEGER'
SMTYPE   = 500; SQLTYPES.SMTYPE   = 'SMALLINT'
DATYPE   = 384; SQLTYPES.DATYPE   = 'DATE'
TITYPE   = 388; SQLTYPES.TITYPE   = 'TIME'
TSTYPE   = 392; SQLTYPES.TSTYPE   = 'TIMESTAMP'

Address TSO "SUBCOM DSNREXX"          /* HOST CMD ENV AVAILABLE? */

IF RC THEN                            /* NO, LET'S MAKE ONE      */
  S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX') /* ADD HOST CMD ENV  */

Address DSNREXX "CONNECT" SSID
If SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL DESCRIBE TABLE :TABLE INTO :SQLDA"

If SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL COMMIT"
Address DSNREXX "DISCONNECT"
If SQLCODE ^= 0 Then Call SQLCA

Select
  When (Left(Type,1) = "S") Then
    Call DrawSelect
  When (Left(Type,1) = "I") Then
    Call DrawInsert
  When (Left(Type,1) = "U") Then
    Call DrawUpdate
  When (Left(Type,1) = "L") Then
    Call DrawLoad
  Otherwise EXIT (20)
End

Do I = LINE.0 To 1 By -1
  LINE = COPIES(" ",LEFTBND-1)||LINE.I
  'ISREDIT LINE_AFTER 'zdest' = DATALINE (Line)'
End
line1 = zdest + 1
'ISREDIT CURSOR = 'line1 0
Exit
```

Figure 245. REXX sample program DRAW (Part 5 of 10)

```
/********************************************************************/
WHEREAMI:; RETURN SIGL
/********************************************************************/
/* Draw SELECT                                                      */
/********************************************************************/
DrawSelect:
  Line.0 = 0
  Line = "SELECT"
  Do I = 1 To SQLDA.SQLD
    If I > 1 Then Line = Line ','
    ColName = '"'SQLDA.I.SQLNAME'"'
    Null = SQLDA.I.SQLTYPE//2
    If Length(Line)+Length(ColName)+LENGTH(" ,") > BndSize THEN
    Do
      L = Line.0 + 1; Line.0 = L
      Line.L = Line
      Line = "        "
    End
    Line = Line ColName
  End I
  If Line ^= "" Then
  Do
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
    Line = "        "
  End
  L = Line.0 + 1; Line.0 = L
  Line.L = "FROM" TABLE
  Return
/********************************************************************/
/* Draw INSERT                                                      */
/********************************************************************/
DrawInsert:
  Line.0 = 0
  Line = "INSERT INTO" TABLE "("
  Do I = 1 To SQLDA.SQLD
    If I > 1 Then Line = Line ','
    ColName = '"'SQLDA.I.SQLNAME'"'
    If Length(Line)+Length(ColName) > BndSize THEN
    Do
      L = Line.0 + 1; Line.0 = L
      Line.L = Line
      Line = "        "
    End
    Line = Line ColName
    If I = SQLDA.SQLD Then Line = Line ')'
  End I
  If Line ^= "" Then
  Do
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
    Line = "        "
  End
```

Figure 245. REXX sample program DRAW (Part 6 of 10)

```
L = Line.0 + 1; Line.0 = L
Line.L = "   VALUES ("
L = Line.0 + 1; Line.0 = L
Line.L = ,
"-- ENTER VALUES BELOW          COLUMN NAME         DATA TYPE"
Do I = 1 To SQLDA.SQLD
  If SQLDA.SQLD > 1 & I < SQLDA.SQLD Then
    Line = "                              , --"
  Else
    Line = "                              ) --"
  Line = Line Left(SQLDA.I.SQLNAME,18)
  Type  = SQLDA.I.SQLTYPE
  Null  = Type//2
  If Null Then Type = Type - 1
  Len   = SQLDA.I.SQLLEN
  Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
  Scale = SQLDA.I.SQLLEN.SQLSCALE
  Select
  When (Type = CHTYPE   ,
       |Type = VCHTYPE  ,
       |Type = LVCHTYPE ,
       |Type = GRTYP    ,
       |Type = VGRTYP   ,
       |Type = LVGRTYP  ) THEN
    Type = SQLTYPES.Type"("STRIP(LEN)")"
  When (Type = FLOTYPE  ) THEN
    Type = SQLTYPES.Type"("STRIP((LEN*4)-11) ")"
  When (Type = DCTYPE   ) THEN
    Type = SQLTYPES.Type"("STRIP(PRCSN)","STRIP(SCALE)")"
  Otherwise
    Type = SQLTYPES.Type
  End
  Line = Line Type
  If Null = 0 Then
  Line = Line "NOT NULL"
  L = Line.0 + 1; Line.0 = L
  Line.L = Line
End I
Return
```

Figure 245. REXX sample program DRAW (Part 7 of 10)

```
/********************************************************************/
/* Draw UPDATE                                                      */
/********************************************************************/
DrawUpdate:
  Line.0 = 1
  Line.1 = "UPDATE" TABLE "SET"
  L = Line.0 + 1; Line.0 = L
  Line.L = ,
  "-- COLUMN NAME           ENTER VALUES BELOW      DATA TYPE"
  Do I = 1 To SQLDA.SQLD
    If I = 1 Then
      Line = "  "
    Else
      Line = " ,"
    Line = Line Left('"'SQLDA.I.SQLNAME'"=',21)
    Line = Line Left(" ",20)
    Type  = SQLDA.I.SQLTYPE
    Null  = Type//2
    If Null Then Type = Type - 1
    Len   = SQLDA.I.SQLLEN
    Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
    Scale = SQLDA.I.SQLLEN.SQLSCALE
    Select
    When (Type = CHTYPE   ,
         |Type = VCHTYPE  ,
         |Type = LVCHTYPE ,
         |Type = GRTYP    ,
         |Type = VGRTYP   ,
         |Type = LVGRTYP  ) THEN
      Type = SQLTYPES.Type"("STRIP(LEN)")"
    When (Type = FLOTYPE  ) THEN
      Type = SQLTYPES.Type"("STRIP((LEN*4)-11) ")"
    When (Type = DCTYPE   ) THEN
      Type = SQLTYPES.Type"("STRIP(PRCSN)","STRIP(SCALE)")"
    Otherwise
      Type = SQLTYPES.Type
    End
    Line = Line "--" Type
    If Null = 0 Then
    Line = Line "NOT NULL"
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
  End I
  L = Line.0 + 1; Line.0 = L
  Line.L = "WHERE"
  Return
```

Figure 245. REXX sample program DRAW (Part 8 of 10)

```
/*******************************************************************/
/* Draw LOAD                                                       */
/*******************************************************************/
DrawLoad:
  Line.0 = 1
  Line.1 = "LOAD DATA INDDN SYSREC INTO TABLE" TABLE
  Position = 1
  Do I = 1 To SQLDA.SQLD
    If I = 1 Then
      Line = " ("
    Else
      Line = " ,"
    Line = Line Left('"'SQLDA.I.SQLNAME'"',20)
    Line = Line "POSITION("RIGHT(POSITION,5)")"
    Type  = SQLDA.I.SQLTYPE
    Null  = Type//2
    If Null Then Type = Type - 1
    Len   = SQLDA.I.SQLLEN
    Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
    Scale = SQLDA.I.SQLLEN.SQLSCALE
    Select
    When (Type = CHTYPE   ,
         |Type = GRTYP    ) THEN
      Type = SQLTYPES.Type"("STRIP(LEN)")"
    When (Type = FLOTYPE  ) THEN
      Type = SQLTYPES.Type"("STRIP((LEN*4)-11) ")"
    When (Type = DCTYPE   ) THEN
    Do
      Type = SQLTYPES.Type "EXTERNAL"
      Type = Type"("STRIP(PRCSN)","STRIP(SCALE)")"
      Len  = (PRCSN+2)%2
    End
    When (Type = DATYPE   ,
         |Type = TITYPE   ,
         |Type = TSTYPE   ) THEN
      Type = SQLTYPES.Type "EXTERNAL"
    Otherwise
      Type = SQLTYPES.Type
    End
    If   (Type = GRTYP    ,
         |Type = VGRTYP   ,
         |Type = LVGRTYP  ) THEN
      Len = Len * 2
    If   (Type = VCHTYPE  ,
         |Type = LVCHTYPE ,
         |Type = VGRTYP   ,
         |Type = LVGRTYP  ) THEN
      Len = Len + 2
    Line = Line Type
    L = Line.0 + 1; Line.0 = L
```

Figure 245. REXX sample program DRAW (Part 9 of 10)

```
      Line.L = Line
      If Null = 1 Then
      Do
        Line = "   "
        Line = Line Left('',20)
        Line = Line "  NULLIF("RIGHT(POSITION,5)")='?'"
        L = Line.0 + 1; Line.0 = L
        Line.L = Line
      End
      Position = Position + Len + 1
    End I
    L = Line.0 + 1; Line.0 = L
    Line.L = " )"
    Return
/**********************************************************************/
/* Display SQLCA                                                      */
/**********************************************************************/
SQLCA:
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLSTATE="SQLSTATE"'"
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLWARN ="SQLWARN.0",",
                ||| SQLWARN.1",",
                ||| SQLWARN.2",",
                ||| SQLWARN.3",",
                ||| SQLWARN.4",",
                ||| SQLWARN.5",",
                ||| SQLWARN.6",",
                ||| SQLWARN.7",",
                ||| SQLWARN.8",",
                ||| SQLWARN.9",",
                ||| SQLWARN.10"'"
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRD ="SQLERRD.1",",
                ||| SQLERRD.2",",
                ||| SQLERRD.3",",
                ||| SQLERRD.4",",
                ||| SQLERRD.5",",
                ||| SQLERRD.6"'"
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRP ="SQLERRP"'"
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRMC ="SQLERRMC"'"
  "ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLCODE ="SQLCODE"'"
  Exit 20
```

Figure 245. REXX sample program DRAW (Part 10 of 10)

Sample COBOL program using DRDA access

The following sample program demonstrates distributed data access using DRDA access.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TWOPHASE.
 AUTHOR.
 REMARKS.
*******************************************************************
*                                                                 *
* MODULE NAME = TWOPHASE                                          *
*                                                                 *
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING                *
*                    TWO PHASE COMMIT AND THE DRDA DISTRIBUTED   *
*                    ACCESS METHOD                               *
*                                                                 *
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989         *
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083          *
*                                                                 *
* STATUS = VERSION 5                                             *
*                                                                 *
* FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS    *
*            USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE    *
*            FROM ONE LOCATION TO ANOTHER.                       *
*                                                                 *
*            NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE     *
*                  TABLE SYSADM.EMP AT LOCATIONS STLEC1 AND      *
*                  STLEC2.                                       *
*                                                                 *
* MODULE TYPE = COBOL PROGRAM                                    *
*    PROCESSOR   = DB2 PRECOMPILER, VS COBOL II                  *
*    MODULE SIZE = SEE LINK EDIT                                 *
*    ATTRIBUTES  = NOT REENTRANT OR REUSABLE                     *
*                                                                 *
* ENTRY POINT =                                                  *
*    PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT                      *
*    LINKAGE = INVOKE FROM DSN RUN                               *
*    INPUT   = NONE                                              *
*    OUTPUT  =                                                   *
*              SYMBOLIC LABEL/NAME = SYSPRINT                    *
*              DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH   *
*                STEP AND THE RESULTANT SQLCA                    *
*                                                                 *
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION            *
*                                                                 *
* EXIT ERROR = NONE                                             *
*                                                                 *
* EXTERNAL REFERENCES =                                         *
*    ROUTINE SERVICES = NONE                                    *
*    DATA-AREAS       = NONE                                    *
*    CONTROL-BLOCKS   =                                         *
*       SQLCA  -   SQL COMMUNICATION AREA                       *
*                                                                 *
* TABLES = NONE                                                 *
*                                                                 *
* CHANGE-ACTIVITY = NONE                                        *
*                                                                 *
*                                                                 *
*                                                                 *
*                                                                 *
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 1 of 8)

```
*  PSEUDOCODE                                            *
*                                                        *
*     MAINLINE.                                          *
*       Perform CONNECT-TO-SITE-1 to establish           *
*         a connection to the local connection.          *
*       If the previous operation was successful Then    *
*         Do.                                             *
*          | Perform PROCESS-CURSOR-SITE-1 to obtain the *
*          |   information about an employee that is      *
*          |   transferring to another location.         *
*          | If the information about the employee was obtained  *
*          |   successfully Then                         *
*          | Do.                                          *
*          |  | Perform UPDATE-ADDRESS to update the information *
*          |  |   to contain current information about the *
*          |  |   employee.                               *
*          |  | Perform CONNECT-TO-SITE-2 to establish    *
*          |  |   a connection to the site where the employee is *
*          |  |   transferring to.                        *
*          |  |   If the connection is established successfully *
*          |  |   Then                                    *
*          |  |     Do.                                   *
*          |  |      | Perform PROCESS-SITE-2 to insert the *
*          |  |      |   employee information at the location *
*          |  |      |   where the employee is transferring to. *
*          |  |     End if the connection was established  *
*          |  |       successfully.                       *
*          | End if the employee information was obtained  *
*          |   successfully.                              *
*       End if the previous operation was successful.     *
*     Perform COMMIT-WORK to COMMIT the changes made to STLEC1 *
*       and STLEC2.                                       *
*                                                        *
*     PROG-END.                                           *
*       Close the printer.                                *
*       Return.                                            *
*                                                        *
*     CONNECT-TO-SITE-1.                                  *
*       Provide a text description of the following step. *
*       Establish a connection to the location where the  *
*         employee is transferring from.                  *
*       Print the SQLCA out.                              *
*                                                        *
*     PROCESS-CURSOR-SITE-1.                              *
*       Provide a text description of the following step. *
*       Open a cursor that will be used to retrieve information *
*         about the transferring employee from this site. *
*       Print the SQLCA out.                              *
*       If the cursor was opened successfully Then        *
*         Do.                                             *
*          | Perform FETCH-DELETE-SITE-1 to retrieve and  *
*          |   delete the information about the transferring *
*          |   employee from this site.                   *
*          | Perform CLOSE-CURSOR-SITE-1 to close the cursor. *
*         End if the cursor was opened successfully.      *
*                                                        *
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 2 of 8)

```
*    FETCH-DELETE-SITE-1.                                        *
*      Provide a text description of the following step.        *
*      Fetch information about the transferring employee.       *
*      Print the SQLCA out.                                     *
*      If the information was retrieved successfully Then       *
*        Do.                                                    *
*        | Perform DELETE-SITE-1 to delete the employee         *
*        |   at this site.                                      *
*        End if the information was retrieved successfully.     *
*                                                               *
*    DELETE-SITE-1.                                             *
*      Provide a text description of the following step.        *
*      Delete the information about the transferring employee   *
*        from this site.                                        *
*      Print the SQLCA out.                                     *
*                                                               *
*    CLOSE-CURSOR-SITE-1.                                       *
*      Provide a text description of the following step.        *
*      Close the cursor used to retrieve information about      *
*        the transferring employee.                             *
*      Print the SQLCA out.                                     *
*                                                               *
*    UPDATE-ADDRESS.                                            *
*      Update the address of the employee.                      *
*      Update the city of the employee.                         *
*      Update the location of the employee.                     *
*                                                               *
*    CONNECT-TO-SITE-2.                                         *
*      Provide a text description of the following step.        *
*      Establish a connection to the location where the         *
*        employee is transferring to.                           *
*      Print the SQLCA out.                                     *
*                                                               *
*    PROCESS-SITE-2.                                            *
*      Provide a text description of the following step.        *
*      Insert the employee information at the location where    *
*        the employee is being transferred to.                  *
*      Print the SQLCA out.                                     *
*                                                               *
*    COMMIT-WORK.                                               *
*      COMMIT all the changes made to STLEC1 and STLEC2.        *
*                                                               *
*****************************************************************


     ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
         SELECT PRINTER, ASSIGN TO S-OUT1.

     DATA DIVISION.
     FILE SECTION.
     FD  PRINTER
         RECORD CONTAINS 120 CHARACTERS
         DATA RECORD IS PRT-TC-RESULTS
         LABEL RECORD IS OMITTED.
     01  PRT-TC-RESULTS.
         03  PRT-BLANK              PIC X(120).
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 3 of 8)

```
      WORKING-STORAGE SECTION.

     ****************************************************************
     * Variable declarations                                       *
     ****************************************************************

      01  H-EMPTBL.
          05  H-EMPNO   PIC X(6).
          05  H-NAME.
              49 H-NAME-LN    PIC S9(4) COMP-4.
              49 H-NAME-DA    PIC X(32).
          05  H-ADDRESS.
              49 H-ADDRESS-LN    PIC S9(4) COMP-4.
              49 H-ADDRESS-DA    PIC X(36).
          05  H-CITY.
              49 H-CITY-LN    PIC S9(4) COMP-4.
              49 H-CITY-DA    PIC X(36).
          05  H-EMPLOC   PIC X(4).
          05  H-SSNO     PIC X(11).
          05  H-BORN     PIC X(10).
          05  H-SEX      PIC X(1).
          05  H-HIRED    PIC X(10).
          05  H-DEPTNO   PIC X(3).
          05  H-JOBCODE  PIC S9(3)V COMP-3.
          05  H-SRATE    PIC S9(5) COMP.
          05  H-EDUC     PIC S9(5) COMP.
          05  H-SAL      PIC S9(6)V9(2) COMP-3.
          05  H-VALIDCHK PIC S9(6)V COMP-3.

       01  H-EMPTBL-IND-TABLE.
           02  H-EMPTBL-IND        PIC S9(4) COMP OCCURS 15 TIMES.

     ****************************************************************
     * Includes for the variables used in the COBOL standard       *
     * language procedures and the SQLCA.                          *
     ****************************************************************

          EXEC SQL INCLUDE COBSVAR END-EXEC.
          EXEC SQL INCLUDE SQLCA END-EXEC.

     ****************************************************************
     * Declaration for the table that contains employee information *
     ****************************************************************

          EXEC SQL DECLARE SYSADM.EMP TABLE
              (EMPNO   CHAR(6) NOT NULL,
               NAME    VARCHAR(32),
               ADDRESS VARCHAR(36) ,
               CITY    VARCHAR(36) ,
               EMPLOC  CHAR(4) NOT NULL,
               SSNO    CHAR(11),
               BORN    DATE,
               SEX     CHAR(1),
               HIRED   CHAR(10),
               DEPTNO  CHAR(3) NOT NULL,
               JOBCODE DECIMAL(3),
               SRATE   SMALLINT,
               EDUC    SMALLINT,
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 4 of 8)

```
             SAL     DECIMAL(8,2) NOT NULL,
             VALCHK  DECIMAL(6))
      END-EXEC.

      *****************************************************************
      * Constants                                                     *
      *****************************************************************

       77  SITE-1                  PIC X(16) VALUE 'STLEC1'.
       77  SITE-2                  PIC X(16) VALUE 'STLEC2'.
       77  TEMP-EMPNO              PIC X(6)  VALUE '080000'.
       77  TEMP-ADDRESS-LN         PIC 99    VALUE 15.
       77  TEMP-CITY-LN            PIC 99    VALUE 18.

      *****************************************************************
      * Declaration of the cursor that will be used to retrieve       *
      * information about a transferring employee                     *
      *****************************************************************

           EXEC SQL DECLARE C1 CURSOR FOR
               SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
                      SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
                      SRATE, EDUC, SAL, VALCHK
               FROM   SYSADM.EMP
               WHERE EMPNO = :TEMP-EMPNO
           END-EXEC.

       PROCEDURE DIVISION.
       A101-HOUSE-KEEPING.
           OPEN OUTPUT PRINTER.

      *****************************************************************
      * An employee is transferring from location STLEC1 to STLEC2.   *
      * Retrieve information about the employee from STLEC1, delete    *
      * the employee from STLEC1 and insert the employee at STLEC2     *
      * using the information obtained from STLEC1.                    *
      *****************************************************************

       MAINLINE.
           PERFORM CONNECT-TO-SITE-1
           IF SQLCODE IS EQUAL TO 0
             PERFORM PROCESS-CURSOR-SITE-1
             IF SQLCODE IS EQUAL TO 0
               PERFORM UPDATE-ADDRESS
               PERFORM CONNECT-TO-SITE-2
               IF SQLCODE IS EQUAL TO 0
                 PERFORM PROCESS-SITE-2.
           PERFORM COMMIT-WORK.
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 5 of 8)

```
      PROG-END.
          CLOSE PRINTER.
          GOBACK.


      ******************************************************************
      * Establish a connection to STLEC1                              *
      ******************************************************************

       CONNECT-TO-SITE-1.

          MOVE 'CONNECT TO STLEC1   ' TO STNAME
          WRITE PRT-TC-RESULTS FROM STNAME
          EXEC SQL
             CONNECT TO :SITE-1
          END-EXEC.
          PERFORM PTSQLCA.


      ******************************************************************
      * Once a connection has been established successfully at STLEC1,*
      * open the cursor that will be used to retrieve information     *
      * about the transferring employee.                             *
      ******************************************************************

       PROCESS-CURSOR-SITE-1.

          MOVE 'OPEN CURSOR C1      ' TO STNAME
          WRITE PRT-TC-RESULTS FROM STNAME
          EXEC SQL
             OPEN C1
          END-EXEC.
          PERFORM PTSQLCA.
          IF SQLCODE IS EQUAL TO ZERO
             PERFORM FETCH-DELETE-SITE-1
             PERFORM CLOSE-CURSOR-SITE-1.


      ******************************************************************
      * Retrieve information about the transferring employee.         *
      * Provided that the employee exists, perform DELETE-SITE-1 to   *
      * delete the employee from STLEC1.                              *
      ******************************************************************

       FETCH-DELETE-SITE-1.

          MOVE 'FETCH C1      ' TO STNAME
          WRITE PRT-TC-RESULTS FROM STNAME
          EXEC SQL
             FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
          END-EXEC.
          PERFORM PTSQLCA.
          IF SQLCODE IS EQUAL TO ZERO
             PERFORM DELETE-SITE-1.
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 6 of 8)

```
                 ******************************************************************
                 * Delete the employee from STLEC1.                               *
                 ******************************************************************

                  DELETE-SITE-1.

                     MOVE 'DELETE EMPLOYEE ' TO STNAME
                     WRITE PRT-TC-RESULTS FROM STNAME
                     MOVE 'DELETE EMPLOYEE     ' TO STNAME
                     EXEC SQL
                        DELETE FROM SYSADM.EMP
                           WHERE EMPNO = :TEMP-EMPNO
                     END-EXEC.
                     PERFORM PTSQLCA.


                 ******************************************************************
                 * Close the cursor used to retrieve information about the        *
                 * transferring employee.                                         *
                 ******************************************************************

                  CLOSE-CURSOR-SITE-1.

                     MOVE 'CLOSE CURSOR C1    ' TO STNAME
                     WRITE PRT-TC-RESULTS FROM STNAME
                     EXEC SQL
                        CLOSE C1
                     END-EXEC.
                     PERFORM PTSQLCA.


                 ******************************************************************
                 * Update certain employee information in order to make it        *
                 * current.                                                       *
                 ******************************************************************

                  UPDATE-ADDRESS.
                     MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
                     MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
                     MOVE TEMP-CITY-LN         TO H-CITY-LN.
                     MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
                     MOVE 'SJCA'               TO H-EMPLOC.


                 ******************************************************************
                 * Establish a connection to STLEC2                               *
                 ******************************************************************

                  CONNECT-TO-SITE-2.

                     MOVE 'CONNECT TO STLEC2   ' TO STNAME
                     WRITE PRT-TC-RESULTS FROM STNAME
                     EXEC SQL
                        CONNECT TO :SITE-2
                     END-EXEC.
                     PERFORM PTSQLCA.
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 7 of 8)

```
      ******************************************************************
      * Using the employee information that was retrieved from STLEC1 *
      * and updated above, insert the employee at STLEC2.             *
      ******************************************************************

       PROCESS-SITE-2.

           MOVE 'INSERT EMPLOYEE     ' TO STNAME
           WRITE PRT-TC-RESULTS FROM STNAME
           EXEC SQL
             INSERT INTO SYSADM.EMP VALUES
             (:H-EMPNO,
              :H-NAME,
              :H-ADDRESS,
              :H-CITY,
              :H-EMPLOC,
              :H-SSNO,
              :H-BORN,
              :H-SEX,
              :H-HIRED,
              :H-DEPTNO,
              :H-JOBCODE,
              :H-SRATE,
              :H-EDUC,
              :H-SAL,
              :H-VALIDCHK)
           END-EXEC.
           PERFORM PTSQLCA.

      ******************************************************************
      * COMMIT any changes that were made at STLEC1 and STLEC2.       *
      ******************************************************************

       COMMIT-WORK.

           MOVE 'COMMIT WORK          ' TO STNAME
           WRITE PRT-TC-RESULTS FROM STNAME
           EXEC SQL
             COMMIT
           END-EXEC.
           PERFORM PTSQLCA.

      ******************************************************************
      * Include COBOL standard language procedures                   *
      ******************************************************************

       INCLUDE-SUBS.
           EXEC SQL INCLUDE COBSSUB END-EXEC.
```

Figure 246. Sample COBOL two-phase commit application for DRDA access (Part 8 of 8)

Sample COBOL program using DB2 private protocol access

The following sample program demonstrates distributed access data using DB2 private protocol access with two-phase commit.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TWOPHASE.
 AUTHOR.
 REMARKS.
 *****************************************************************
 *                                                               *
 * MODULE NAME = TWOPHASE                                         *
 *                                                               *
 * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING               *
 *                    TWO PHASE COMMIT AND DB2 PRIVATE PROTOCOL   *
 *                    DISTRIBUTED ACCESS METHOD                   *
 *                                                               *
 * COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989        *
 * REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083          *
 *                                                               *
 * STATUS = VERSION 5                                            *
 *                                                               *
 * FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS   *
 *            USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE   *
 *            FROM ONE LOCATION TO ANOTHER.                      *
 *                                                               *
 *            NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE    *
 *                  TABLE SYSADM.EMP AT LOCATIONS STLEC1 AND     *
 *                  STLEC2.                                      *
 *                                                               *
 * MODULE TYPE = COBOL PROGRAM                                   *
 *    PROCESSOR  = DB2 PRECOMPILER, VS COBOL II                  *
 *    MODULE SIZE = SEE LINK EDIT                                *
 *    ATTRIBUTES  = NOT REENTRANT OR REUSABLE                    *
 *                                                               *
 * ENTRY POINT =                                                 *
 *    PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT                     *
 *    LINKAGE = INVOKE FROM DSN RUN                              *
 *    INPUT   = NONE                                             *
 *    OUTPUT  =                                                  *
 *              SYMBOLIC LABEL/NAME = SYSPRINT                   *
 *              DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH  *
 *                STEP AND THE RESULTANT SQLCA                   *
 *                                                               *
 * EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION            *
 *                                                               *
 * EXIT ERROR = NONE                                            *
 *                                                               *
 * EXTERNAL REFERENCES =                                         *
 *    ROUTINE SERVICES = NONE                                    *
 *    DATA-AREAS       = NONE                                    *
 *    CONTROL-BLOCKS   =                                         *
 *       SQLCA  -  SQL COMMUNICATION AREA                        *
 *                                                               *
 * TABLES = NONE                                                 *
 *                                                               *
 * CHANGE-ACTIVITY = NONE                                        *
 *                                                               *
 *                                                               *
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access (Part 1 of 7)

```
*                                                              *
* PSEUDOCODE                                                   *
*                                                              *
*    MAINLINE.                                                 *
*      Perform PROCESS-CURSOR-SITE-1 to obtain the information *
*        about an employee that is transferring to another    *
*        location.                                            *
*      If the information about the employee was obtained      *
*        successfully Then                                     *
*        Do.                                                  *
*        │ Perform UPDATE-ADDRESS to update the information to *
*        │   contain current information about the employee.   *
*        │ Perform PROCESS-SITE-2 to insert the employee       *
*        │   information at the location where the employee is *
*        │   transferring to.                                 *
*        End if the employee information was obtained          *
*          successfully.                                      *
*      Perform COMMIT-WORK to COMMIT the changes made to STLEC1 *
*        and STLEC2.                                          *
*                                                              *
*    PROG-END.                                                 *
*      Close the printer.                                     *
*      Return.                                                *
*                                                              *
*    PROCESS-CURSOR-SITE-1.                                   *
*      Provide a text description of the following step.      *
*      Open a cursor that will be used to retrieve information *
*        about the transferring employee from this site.      *
*      Print the SQLCA out.                                   *
*      If the cursor was opened successfully Then             *
*        Do.                                                  *
*        │ Perform FETCH-DELETE-SITE-1 to retrieve and        *
*        │   delete the information about the transferring     *
*        │   employee from this site.                        *
*        │ Perform CLOSE-CURSOR-SITE-1 to close the cursor.    *
*        End if the cursor was opened successfully.           *
*                                                              *
*    FETCH-DELETE-SITE-1.                                     *
*      Provide a text description of the following step.      *
*      Fetch information about the transferring employee.     *
*      Print the SQLCA out.                                   *
*      If the information was retrieved successfully Then     *
*        Do.                                                  *
*        │ Perform DELETE-SITE-1 to delete the employee       *
*        │   at this site.                                   *
*        End if the information was retrieved successfully.   *
*                                                              *
*    DELETE-SITE-1.                                           *
*      Provide a text description of the following step.      *
*      Delete the information about the transferring employee *
*        from this site.                                      *
*      Print the SQLCA out.                                   *
*                                                              *
*    CLOSE-CURSOR-SITE-1.                                     *
*      Provide a text description of the following step.      *
*      Close the cursor used to retrieve information about    *
*        the transferring employee.                          *
*      Print the SQLCA out.                                   *
*                                                              *
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access (Part 2 of 7)

```
*   UPDATE-ADDRESS.                                      *
*      Update the address of the employee.              *
*      Update the city of the employee.                 *
*      Update the location of the employee.             *
*                                                        *
*   PROCESS-SITE-2.                                      *
*      Provide a text description of the following step. *
*      Insert the employee information at the location where *
*        the employee is being transferred to.          *
*      Print the SQLCA out.                              *
*                                                        *
*   COMMIT-WORK.                                         *
*      COMMIT all the changes made to STLEC1 and STLEC2. *
*                                                        *
*********************************************************************


 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINTER, ASSIGN TO S-OUT1.

 DATA DIVISION.
 FILE SECTION.
 FD  PRINTER
     RECORD CONTAINS 120 CHARACTERS
     DATA RECORD IS PRT-TC-RESULTS
     LABEL RECORD IS OMITTED.
 01  PRT-TC-RESULTS.
     03  PRT-BLANK              PIC X(120).

 WORKING-STORAGE SECTION.

 *****************************************************************
 * Variable declarations                                        *
 *****************************************************************

 01  H-EMPTBL.
     05  H-EMPNO   PIC X(6).
     05  H-NAME.
         49 H-NAME-LN   PIC S9(4) COMP-4.
         49 H-NAME-DA   PIC X(32).
     05  H-ADDRESS.
         49 H-ADDRESS-LN   PIC S9(4) COMP-4.
         49 H-ADDRESS-DA   PIC X(36).
     05  H-CITY.
         49 H-CITY-LN   PIC S9(4) COMP-4.
         49 H-CITY-DA   PIC X(36).
     05  H-EMPLOC   PIC X(4).
     05  H-SSNO     PIC X(11).
     05  H-BORN     PIC X(10).
     05  H-SEX      PIC X(1).
     05  H-HIRED    PIC X(10).
     05  H-DEPTNO   PIC X(3).
     05  H-JOBCODE  PIC S9(3)V COMP-3.
     05  H-SRATE    PIC S9(5) COMP.
     05  H-EDUC     PIC S9(5) COMP.
     05  H-SAL      PIC S9(6)V9(2) COMP-3.
     05  H-VALIDCHK PIC S9(6)V COMP-3.
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access
(Part 3 of 7)

```
 01  H-EMPTBL-IND-TABLE.
     02  H-EMPTBL-IND        PIC S9(4) COMP OCCURS 15 TIMES.


*****************************************************************
* Includes for the variables used in the COBOL standard        *
* language procedures and the SQLCA.                           *
*****************************************************************

     EXEC SQL INCLUDE COBSVAR END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.


*****************************************************************
* Declaration for the table that contains employee information *
*****************************************************************

     EXEC SQL DECLARE SYSADM.EMP TABLE
         (EMPNO   CHAR(6) NOT NULL,
          NAME    VARCHAR(32),
          ADDRESS VARCHAR(36) ,
          CITY    VARCHAR(36) ,
          EMPLOC  CHAR(4) NOT NULL,
          SSNO    CHAR(11),
          BORN    DATE,
          SEX     CHAR(1),
          HIRED   CHAR(10),
          DEPTNO  CHAR(3) NOT NULL,
          JOBCODE DECIMAL(3),
          SRATE   SMALLINT,
          EDUC    SMALLINT,
          SAL     DECIMAL(8,2) NOT NULL,
          VALCHK  DECIMAL(6))
     END-EXEC.


*****************************************************************
* Constants                                                    *
*****************************************************************

 77  TEMP-EMPNO               PIC X(6)  VALUE '080000'.
 77  TEMP-ADDRESS-LN          PIC 99    VALUE 15.
 77  TEMP-CITY-LN             PIC 99    VALUE 18.


*****************************************************************
* Declaration of the cursor that will be used to retrieve      *
* information about a transferring employee                    *
*****************************************************************

     EXEC SQL DECLARE C1 CURSOR FOR
         SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
                SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
                SRATE, EDUC, SAL, VALCHK
         FROM   STLEC1.SYSADM.EMP
         WHERE EMPNO = :TEMP-EMPNO
     END-EXEC.
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access (Part 4 of 7)

```
          PROCEDURE DIVISION.
          A101-HOUSE-KEEPING.
              OPEN OUTPUT PRINTER.

          ******************************************************************
          * An employee is transferring from location STLEC1 to STLEC2.   *
          * Retrieve information about the employee from STLEC1, delete    *
          * the employee from STLEC1 and insert the employee at STLEC2     *
          * using the information obtained from STLEC1.                    *
          ******************************************************************

          MAINLINE.
              PERFORM PROCESS-CURSOR-SITE-1
              IF SQLCODE IS EQUAL TO 0
                 PERFORM UPDATE-ADDRESS
                 PERFORM PROCESS-SITE-2.
              PERFORM COMMIT-WORK.

          PROG-END.
              CLOSE PRINTER.
              GOBACK.

          ******************************************************************
          * Open the cursor that will be used to retrieve information      *
          * about the transferring employee.                              *
          ******************************************************************

          PROCESS-CURSOR-SITE-1.

              MOVE 'OPEN CURSOR C1      ' TO STNAME
              WRITE PRT-TC-RESULTS FROM STNAME
              EXEC SQL
                 OPEN C1
              END-EXEC.
              PERFORM PTSQLCA.
              IF SQLCODE IS EQUAL TO ZERO
                  PERFORM FETCH-DELETE-SITE-1
                  PERFORM CLOSE-CURSOR-SITE-1.

          ******************************************************************
          * Retrieve information about the transferring employee.         *
          * Provided that the employee exists, perform DELETE-SITE-1 to    *
          * delete the employee from STLEC1.                              *
          ******************************************************************

          FETCH-DELETE-SITE-1.

              MOVE 'FETCH C1       ' TO STNAME
              WRITE PRT-TC-RESULTS FROM STNAME
              EXEC SQL
                 FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
              END-EXEC.
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access
(Part 5 of 7)

```
            PERFORM PTSQLCA.
            IF SQLCODE IS EQUAL TO ZERO
                PERFORM DELETE-SITE-1.


       ******************************************************************
       * Delete the employee from STLEC1.                              *
       ******************************************************************

        DELETE-SITE-1.

            MOVE 'DELETE EMPLOYEE ' TO STNAME
            WRITE PRT-TC-RESULTS FROM STNAME
            MOVE 'DELETE EMPLOYEE      ' TO STNAME
            EXEC SQL
               DELETE FROM STLEC1.SYSADM.EMP
                  WHERE EMPNO = :TEMP-EMPNO
            END-EXEC.
            PERFORM PTSQLCA.


       ******************************************************************
       * Close the cursor used to retrieve information about the       *
       * transferring employee.                                        *
       ******************************************************************

        CLOSE-CURSOR-SITE-1.

            MOVE 'CLOSE CURSOR C1     ' TO STNAME
            WRITE PRT-TC-RESULTS FROM STNAME
            EXEC SQL
               CLOSE C1
            END-EXEC.
            PERFORM PTSQLCA.


       ******************************************************************
       * Update certain employee information in order to make it       *
       * current.                                                      *
       ******************************************************************

        UPDATE-ADDRESS.
            MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
            MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
            MOVE TEMP-CITY-LN         TO H-CITY-LN.
            MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
            MOVE 'SJCA'               TO H-EMPLOC.
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access (Part 6 of 7)

```
      *******************************************************************
      * Using the employee information that was retrieved from STLEC1 *
      * and updated above, insert the employee at STLEC2.            *
      *******************************************************************

       PROCESS-SITE-2.

           MOVE 'INSERT EMPLOYEE     ' TO STNAME
           WRITE PRT-TC-RESULTS FROM STNAME
           EXEC SQL
              INSERT INTO STLEC2.SYSADM.EMP VALUES
              (:H-EMPNO,
               :H-NAME,
               :H-ADDRESS,
               :H-CITY,
               :H-EMPLOC,
               :H-SSNO,
               :H-BORN,
               :H-SEX,
               :H-HIRED,
               :H-DEPTNO,
               :H-JOBCODE,
               :H-SRATE,
               :H-EDUC,
               :H-SAL,
               :H-VALIDCHK)
           END-EXEC.
           PERFORM PTSQLCA.

      *******************************************************************
      * COMMIT any changes that were made at STLEC1 and STLEC2.       *
      *******************************************************************

       COMMIT-WORK.

           MOVE 'COMMIT WORK          ' TO STNAME
           WRITE PRT-TC-RESULTS FROM STNAME
           EXEC SQL
              COMMIT
           END-EXEC.
           PERFORM PTSQLCA.

      *******************************************************************
      * Include COBOL standard language procedures                   *
      *******************************************************************

       INCLUDE-SUBS.
           EXEC SQL INCLUDE COBSSUB END-EXEC.
```

Figure 247. Sample COBOL two-phase commit application for DB2 private protocol access (Part 7 of 7)

Examples of using stored procedures

This section contains sample programs that you can refer to when programming your stored procedure applications. DSN710.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Calling a stored procedure from a C program

This example shows how to call the C language version of the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention. Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  /***********************************************************/
  /* Include the SQLCA and SQLDA                            */
  /***********************************************************/
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL INCLUDE SQLDA;
  /***********************************************************/
  /* Declare variables that are not SQL-related.            */
  /***********************************************************/
  short int i;              /* Loop counter                 */
  /***********************************************************/
  /* Declare the following:                                 */
  /* - Parameters used to call stored procedure GETPRML     */
  /* - An SQLDA for DESCRIBE PROCEDURE                      */
  /* - An SQLDA for DESCRIBE CURSOR                         */
  /* - Result set variable locators for up to three result */
  /*   sets                                                 */
  /***********************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char procnm[19];        /* INPUT parm -- PROCEDURE name */
    char schema[9];         /* INPUT parm -- User's schema  */
    long int out_code;      /* OUTPUT -- SQLCODE from the   */
                            /*   SELECT operation.          */
    struct {
         short int parmlen;
         char parmtxt[254];
       } parmlst;           /* OUTPUT -- RUNOPTS values     */
                            /*   for the matching row in    */
                            /*   catalog table SYSROUTINES  */
    struct indicators {
      short int procnm_ind;
      short int schema_ind;
      short int out_code_ind;
      short int parmlst_ind;
      } parmind;
                            /* Indicator variable structure */

    struct sqlda *proc_da;
                            /* SQLDA for DESCRIBE PROCEDURE */
    struct sqlda *res_da;
                            /* SQLDA for DESCRIBE CURSOR    */
    static volatile
      SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
                            /* Locator variables            */
  EXEC SQL END DECLARE SECTION;
```

Figure 248. Calling a stored procedure from a C program (Part 1 of 4)

```
/*************************************************************/
/* Allocate the SQLDAs to be used for DESCRIBE              */
/* PROCEDURE and DESCRIBE CURSOR.  Assume that at most      */
/* three cursors are returned and that each result set      */
/* has no more than five columns.                           */
/*************************************************************/
proc_da = (struct sqlda *)malloc(SQLDASIZE(3));
res_da = (struct sqlda *)malloc(SQLDASIZE(5));


  /*************************************************************/
  /* Call the GETPRML stored procedure to retrieve the        */
  /* RUNOPTS values for the stored procedure.   In this       */
  /* example, we request the PARMLIST definition for the      */
  /* stored procedure named DSN8EP2.                          */
  /*                                                          */
  /* The call should complete with SQLCODE +466 because       */
  /* GETPRML returns result sets.                             */
  /*************************************************************/
  strcpy(procnm,"dsn8ep2            ");
              /* Input parameter -- PROCEDURE to be found  */
  strcpy(schema,"        ");
              /* Input parameter -- Schema name for proc   */
  parmind.procnm_ind=0;
  parmind.schema_ind=0;
  parmind.out_code_ind=0;
              /* Indicate that none of the input parameters */
              /* have null values                          */
  parmind.parmlst_ind=-1;
              /* The parmlst parameter is an output parm.  */
              /* Mark PARMLST parameter as null, so the DB2 */
              /* requester doesn't have to send the entire  */
              /* PARMLST variable to the server.  This      */
              /* helps reduce network I/O time, because     */
              /* PARMLST is fairly large.                   */
  EXEC SQL
    CALL GETPRML(:procnm INDICATOR :parmind.procnm_ind,
                 :schema INDICATOR :parmind.schema_ind,
                 :out_code INDICATOR :parmind.out_code_ind,
                 :parmlst INDICATOR :parmind.parmlst_ind);
  if(SQLCODE!=+466)         /* If SQL CALL failed,           */
  {
                            /*    print the SQLCODE and any  */
                            /*    message tokens             */
      printf("SQL CALL failed due to SQLCODE = %d\n",SQLCODE);
      printf("sqlca.sqlerrmc = ");
      for(i=0;i<sqlca.sqlerrml;i++)
        printf("%c",sqlca.sqlerrmc[i]);
      printf("\n");
  }
```

Figure 248. Calling a stored procedure from a C program (Part 2 of 4)

```
            else                    /* If the CALL worked,          */
              if(out_code!=0)       /* Did GETPRML hit an error?    */
                printf("GETPRML failed due to RC = %d\n",out_code);
              /**********************************************************/
              /* If everything worked, do the following:              */
              /* - Print out the parameters returned.                 */
              /* - Retrieve the result sets returned.                 */
              /**********************************************************/
              else
              {
                printf("RUNOPTS = %s\n",parmlst.parmtxt);
                                      /* Print out the runopts list   */

                /**********************************************************/
                /* Use the statement DESCRIBE PROCEDURE to              */
                /* return information about the result sets in the      */
                /* SQLDA pointed to by proc_da:                         */
                /* - SQLD contains the number of result sets that were  */
                /*   returned by the stored procedure.                  */
                /* - Each SQLVAR entry has the following information     */
                /*   about a result set:                                */
                /*   - SQLNAME contains the name of the cursor that     */
                /*     the stored procedure uses to return the result   */
                /*     set.                                             */
                /*   - SQLIND contains an estimate of the number of     */
                /*     rows in the result set.                          */
                /*   - SQLDATA contains the result locator value for    */
                /*     the result set.                                  */
                /**********************************************************/
                EXEC SQL DESCRIBE PROCEDURE INTO :*proc_da;
                /**********************************************************/
                /* Assume that you have examined SQLD and determined    */
                /* that there is one result set.  Use the statement     */
                /* ASSOCIATE LOCATORS to establish a result set locator */
                /* for the result set.                                  */
                /**********************************************************/
                EXEC SQL ASSOCIATE LOCATORS (:loc1) WITH PROCEDURE GETPRML;

                /**********************************************************/
                /* Use the statement ALLOCATE CURSOR to associate a     */
                /* cursor for the result set.                           */
                /**********************************************************/
                EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
                /**********************************************************/
                /* Use the statement DESRIBE CURSOR to determine the    */
                /* columns in the result set.                           */
                /**********************************************************/
                EXEC SQL DESCRIBE CURSOR C1 INTO :*res_da;
```

Figure 248. Calling a stored procedure from a C program (Part 3 of 4)

```
/********************************************************/
/* Call a routine (not shown here) to do the following: */
/* - Allocate a buffer for data and indicator values    */
/*   fetched from the result table.                     */
/* - Update the SQLDATA and SQLIND fields in each        */
/*   SQLVAR of *res_da with the addresses at which to    */
/*   to put the fetched data and values of indicator     */
/*   variables.                                          */
/********************************************************/
alloc_outbuff(res_da);

/********************************************************/
/* Fetch the data from the result table.                */
/********************************************************/
while(SQLCODE==0)
   EXEC SQL FETCH C1 USING DESCRIPTOR :*res_da;
   }
  return;
 }
```

Figure 248. Calling a stored procedure from a C program (Part 4 of 4)

Calling a stored procedure from a COBOL program

This example shows how to call a version of the GETPRML stored procedure that uses the GENERAL linkage convention from a COBOL program on an MVS system. Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID.    CALPRML.

        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT REPOUT
                  ASSIGN TO UT-S-SYSPRINT.

        DATA DIVISION.
        FILE SECTION.
        FD  REPOUT
                RECORD CONTAINS 127 CHARACTERS
                LABEL RECORDS ARE OMITTED
                DATA RECORD IS REPREC.
        01  REPREC                    PIC X(127).

        WORKING-STORAGE SECTION.
        ****************************************************
        * MESSAGES FOR SQL CALL                           *
        ****************************************************
        01  SQLREC.
                02  BADMSG    PIC X(34) VALUE
                      ' SQL CALL FAILED DUE TO SQLCODE = '.
                02  BADCODE   PIC +9(5) USAGE DISPLAY.
                02  FILLER    PIC X(80) VALUE SPACES.
        01  ERRMREC.
                02  ERRMMSG   PIC X(12) VALUE ' SQLERRMC = '.
                02  ERRMCODE  PIC X(70).
                02  FILLER    PIC X(38) VALUE SPACES.
        01  CALLREC.
                02  CALLMSG   PIC X(28) VALUE
                      ' GETPRML FAILED DUE TO RC = '.
                02  CALLCODE  PIC +9(5) USAGE DISPLAY.
                02  FILLER    PIC X(42) VALUE SPACES.
        01  RSLTREC.
                02  RSLTMSG   PIC X(15) VALUE
                      ' TABLE NAME IS '.
                02  TBLNAME   PIC X(18) VALUE SPACES.
                02  FILLER    PIC X(87) VALUE SPACES.
```

Figure 249. Calling a stored procedure from a COBOL program (Part 1 of 3)

```
     ****************************************************
     * WORK AREAS                                       *
     ****************************************************
      01  PROCNM                 PIC X(18).
      01  SCHEMA                 PIC X(8).
      01  OUT-CODE               PIC S9(9) USAGE COMP.
      01  PARMLST.
          49 PARMLEN       PIC S9(4) USAGE COMP.
          49 PARMTXT       PIC X(254).
      01  PARMBUF REDEFINES PARMLST.
          49 PARBLEN       PIC S9(4) USAGE COMP.
          49 PARMARRY      PIC X(127) OCCURS 2 TIMES.
      01  NAME.
          49 NAMELEN       PIC S9(4) USAGE COMP.
          49 NAMETXT       PIC X(18).
      77  PARMIND          PIC S9(4) COMP.
      77  I                PIC S9(4) COMP.
      77  NUMLINES         PIC S9(4) COMP.
     ****************************************************
     * DECLARE A RESULT SET LOCATOR FOR THE RESULT SET  *
     * THAT IS RETURNED.                                *
     ****************************************************
      01  LOC              USAGE SQL TYPE IS
                           RESULT-SET-LOCATOR VARYING.


     ****************************************************
     * SQL INCLUDE FOR SQLCA                            *
     ****************************************************
          EXEC SQL INCLUDE SQLCA  END-EXEC.

      PROCEDURE DIVISION.
     *------------------
      PROG-START.
           OPEN OUTPUT REPOUT.
     *             OPEN OUTPUT FILE
           MOVE 'DSN8EP2          ' TO PROCNM.
     *             INPUT PARAMETER -- PROCEDURE TO BE FOUND
           MOVE SPACES TO SCHEMA.
     *             INPUT PARAMETER -- SCHEMA IN SYSROUTINES
           MOVE -1 TO PARMIND.
     *             THE PARMLST PARAMETER IS AN OUTPUT PARM.
     *             MARK PARMLST PARAMETER AS NULL, SO THE DB2
     *             REQUESTER DOESN'T HAVE TO SEND THE ENTIRE
     *             PARMLST VARIABLE TO THE SERVER.  THIS
     *             HELPS REDUCE NETWORK I/O TIME, BECAUSE
     *             PARMLST IS FAIRLY LARGE.
          EXEC SQL
             CALL GETPRML(:PROCNM,
                          :SCHEMA,
                          :OUT-CODE,
                          :PARMLST INDICATOR :PARMIND)
          END-EXEC.
```

Figure 249. Calling a stored procedure from a COBOL program (Part 2 of 3)

```
*                   MAKE THE CALL
          IF SQLCODE NOT EQUAL TO +466 THEN
*                       IF CALL RETURNED BAD SQLCODE
            MOVE SQLCODE TO BADCODE
            WRITE REPREC FROM SQLREC
            MOVE SQLERRMC TO ERRMCODE
            WRITE REPREC FROM ERRMREC
          ELSE
            PERFORM GET-PARMS
            PERFORM GET-RESULT-SET.
 PROG-END.
        CLOSE REPOUT.
*                   CLOSE OUTPUT FILE
        GOBACK.
 PARMPRT.
        MOVE SPACES TO REPREC.
        WRITE REPREC FROM PARMARRY(I)
            AFTER ADVANCING 1 LINE.
 GET-PARMS.
*                   IF THE CALL WORKED,
      IF OUT-CODE NOT EQUAL TO 0 THEN
*                   DID GETPRML HIT AN ERROR?
        MOVE OUT-CODE TO CALLCODE
        WRITE REPREC FROM CALLREC
      ELSE
*                   EVERYTHING WORKED
        DIVIDE 127 INTO PARMLEN GIVING NUMLINES ROUNDED
*                   FIND OUT HOW MANY LINES TO PRINT
        PERFORM PARMPRT VARYING I
          FROM 1 BY 1 UNTIL I GREATER THAN NUMLINES.
 GET-RESULT-SET.
*****************************************************
* ASSUME YOU KNOW THAT ONE RESULT SET IS RETURNED,  *
* AND YOU KNOW THE FORMAT OF THAT RESULT SET.       *
* ALLOCATE A CURSOR FOR THE RESULT SET, AND FETCH   *
* THE CONTENTS OF THE RESULT SET.                   *
*****************************************************
      EXEC SQL ASSOCIATE LOCATORS (:LOC)
        WITH PROCEDURE GETPRML
      END-EXEC.
*                   LINK THE RESULT SET TO THE LOCATOR
      EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC
      END-EXEC.
*                   LINK THE CURSOR TO THE RESULT SET
      PERFORM GET-ROWS VARYING I
        FROM 1 BY 1 UNTIL SQLCODE EQUAL TO +100.
 GET-ROWS.
      EXEC SQL FETCH C1 INTO :NAME
      END-EXEC.
     MOVE NAME TO TBLNAME.
     WRITE REPREC FROM RSLTREC
       AFTER ADVANCING 1 LINE.
```

Figure 249. Calling a stored procedure from a COBOL program (Part 3 of 3)

Calling a stored procedure from a PL/I program

This example shows how to call a version of the GETPRML stored procedure that
uses the GENERAL linkage convention from a PL/I program on an MVS system.

```
*PROCESS SYSTEM(MVS);
 CALPRML:
   PROC OPTIONS(MAIN);

   /************************************************************/
   /* Declare the parameters used to call the GETPRML         */
   /* stored procedure.                                       */
   /************************************************************/
   DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
           SCHEMA CHAR(8),     /* INPUT parm -- User's schema  */
           OUT_CODE FIXED BIN(31),
                               /* OUTPUT -- SQLCODE from the   */
                               /*   SELECT operation.          */
           PARMLST CHAR(254)   /* OUTPUT -- RUNOPTS for        */
                   VARYING,    /*   the matching row in the    */
                               /*   catalog table SYSROUTINES  */
           PARMIND FIXED BIN(15);
                               /* PARMLST indicator variable   */
   /************************************************************/
   /* Include the SQLCA                                       */
   /************************************************************/
   EXEC SQL INCLUDE SQLCA;
   /************************************************************/
   /* Call the GETPRML stored procedure to retrieve the       */
   /* RUNOPTS values for the stored procedure.   In this      */
   /* example, we request the RUNOPTS values for the          */
   /* stored procedure named DSN8EP2.                         */
   /************************************************************/
   PROCNM = 'DSN8EP2';
               /* Input parameter -- PROCEDURE to be found    */
   SCHEMA = ' ';
               /* Input parameter -- SCHEMA in SYSROUTINES    */
   PARMIND = -1; /* The PARMLST parameter is an output parm.  */
               /* Mark PARMLST parameter as null, so the DB2  */
               /* requester doesn't have to send the entire   */
               /* PARMLST variable to the server.  This       */
               /* helps reduce network I/O time, because      */
               /* PARMLST is fairly large.                    */
   EXEC SQL
     CALL GETPRML(:PROCNM,
                  :SCHEMA,
                  :OUT_CODE,
                  :PARMLST INDICATOR :PARMIND);
```

Figure 250. Calling a stored procedure from a PL/I program (Part 1 of 2)

```
     IF SQLCODE¬=0 THEN          /* If SQL CALL failed,          */
       DO;
         PUT SKIP EDIT('SQL CALL failed due to SQLCODE = ',
             SQLCODE) (A(34),A(14));
         PUT SKIP EDIT('SQLERRM = ',
             SQLERRM) (A(10),A(70));
       END;
     ELSE                        /* If the CALL worked,         */
       IF OUT_CODE¬=0 THEN       /* Did GETPRML hit an error?   */
         PUT SKIP EDIT('GETPRML failed due to RC = ',
             OUT_CODE) (A(33),A(14));
       ELSE                      /* Everything worked.          */
         PUT SKIP EDIT('RUNOPTS = ', PARMLST) (A(11),A(200));
     RETURN;
  END CALPRML;
```

Figure 250. Calling a stored procedure from a PL/I program (Part 2 of 2)

C stored procedure: GENERAL

This example stored procedure does the following:

- Searches the DB2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention used for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT statement and the value of the RUNOPTS column from SYSROUTINES.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE C
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT SAMPPROG
  PROGRAM TYPE MAIN
  SECURITY DB2
  RESULT SETS 2
  COMMIT ON RETURN NO;
```

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/**************************************************************/
/* Declare C variables for SQL operations on the parameters.  */
/* These are local variables to the C program, which you must */
/* copy to and from the parameter list provided to the stored */
/* procedure.                                                 */
/**************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char SCHEMA[9];
char PARMLST[255];
EXEC SQL END DECLARE SECTION;

/**************************************************************/
/* Declare cursors for returning result sets to the caller.   */
/**************************************************************/
  EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:SCHEMA;

main(argc,argv)
  int argc;
  char *argv[];
  {
      /********************************************************/
      /* Copy the input parameters into the area reserved in  */
      /* the program for SQL processing.                      */
      /********************************************************/
  strcpy(PROCNM, argv[1]);
  strcpy(SCHEMA, argv[2]);

      /********************************************************/
      /* Issue the SQL SELECT against the SYSROUTINES          */
      /* DB2 catalog table.                                   */
      /********************************************************/
  strcpy(PARMLST, "");               /* Clear PARMLST          */
  EXEC SQL
    SELECT RUNOPTS INTO :PARMLST
        FROM SYSIBM.ROUTINES
        WHERE NAME=:PROCNM AND
            SCHEMA=:SCHEMA;
```

Figure 251. A C stored procedure with linkage convention GENERAL (Part 1 of 2)

```
      /*********************************************************/
      /* Copy SQLCODE to the output parameter list.            */
      /*********************************************************/
  *(int *) argv[3] = SQLCODE;

      /*********************************************************/
      /* Copy the PARMLST value returned by the SELECT back to*/
      /* the parameter list provided to this stored procedure.*/
      /*********************************************************/
  strcpy(argv[4], PARMLST);

      /*********************************************************/
      /* Open cursor C1 to cause DB2 to return a result set    */
      /* to the caller.                                        */
      /*********************************************************/
  EXEC SQL OPEN C1;
 }
```

Figure 251. A C stored procedure with linkage convention GENERAL (Part 2 of 2)

C stored procedure: GENERAL WITH NULLS

This example stored procedure does the following:

- Searches the DB2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE C
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL WITH NULLS
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT SAMPPROG
  PROGRAM TYPE MAIN
  SECURITY DB2
  RESULT SETS 2
  COMMIT ON RETURN NO;
```

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/***************************************************************/
/* Declare C variables used for SQL operations on the         */
/* parameters. These are local variables to the C program,    */
/* which you must copy to and from the parameter list provided */
/* to the stored procedure.                                   */
/***************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char SCHEMA[9];
char PARMLST[255];
struct INDICATORS {
  short int PROCNM_IND;
  short int SCHEMA_IND;
  short int OUT_CODE_IND;
  short int PARMLST_IND;
 } PARM_IND;
EXEC SQL END DECLARE SECTION;

/***************************************************************/
/* Declare cursors for returning result sets to the caller.   */
/***************************************************************/
  EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:SCHEMA;

main(argc,argv)
  int argc;
  char *argv[];
 {

      /*******************************************************/
      /* Copy the input parameters into the area reserved in */
      /* the local program for SQL processing.               */
      /*******************************************************/
  strcpy(PROCNM, argv[1]);
  strcpy(SCHEMA, argv[2]);

      /*******************************************************/
      /* Copy null indicator values for the parameter list.  */
      /*******************************************************/
  memcpy(&PARM_IND,(struct INDICATORS *) argv[5],
    sizeof(PARM_IND));
```

Figure 252. A C stored procedure with linkage convention GENERAL WITH NULLS (Part 1 of 2)

```
                /*****************************************************/
                /* If any input parameter is NULL, return an error   */
                /* return code and assign a NULL value to PARMLST.    */
                /*****************************************************/
     if (PARM_IND.PROCNM_IND<0 ||
         PARM_IND.SCHEMA_IND<0 || {
       *(int *) argv[3] = 9999;           /* set output return code  */
       PARM_IND.OUT_CODE_IND = 0;         /* value is not NULL       */
       PARM_IND.PARMLST_IND = -1;         /* PARMLST is NULL         */
       }

     else {
                /*****************************************************/
                /* If the input parameters are not NULL, issue the SQL */
                /* SELECT against the SYSIBM.SYSROUTINES catalog     */
                /* table.                                            */
                /*****************************************************/
         strcpy(PARMLST, "");          /* Clear PARMLST              */
         EXEC SQL
           SELECT RUNOPTS INTO :PARMLST
               FROM SYSIBM.SYSROUTINES
               WHERE NAME=:PROCNM AND
                     SCHEMA=:SCHEMA;
                /*****************************************************/
                /* Copy SQLCODE to the output parameter list.        */
                /*****************************************************/
         *(int *) argv[3] = SQLCODE;
         PARM_IND.OUT_CODE_IND = 0;        /* OUT_CODE is not NULL */
       }

                /*****************************************************/
                /* Copy the RUNOPTS value back to the output parameter */
                /* area.                                             */
                /*****************************************************/
       strcpy(argv[4], PARMLST);

                /*****************************************************/
                /* Copy the null indicators back to the output parameter*/
                /* area.                                             */
                /*****************************************************/
       memcpy((struct INDICATORS *) argv[5],&PARM_IND,
         sizeof(PARM_IND));

                /*****************************************************/
                /* Open cursor C1 to cause DB2 to return a result set */
                /* to the caller.                                    */
                /*****************************************************/
       EXEC SQL OPEN C1;
       }
```

Figure 252. A C stored procedure with linkage convention GENERAL WITH NULLS (Part 2 of 2)

COBOL stored procedure: GENERAL

This example stored procedure does the following:

- Searches the catalog table SYSROUTINES for a row matching the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

This stored procedure is able to return a NULL value for the output host variables.

The linkage convention for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE COBOL
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT SAMPPROG
  PROGRAM TYPE MAIN
  SECURITY DB2
  RESULT SETS 2
  COMMIT ON RETURN NO;
```

```
                    CBL RENT
                    IDENTIFICATION DIVISION.
                    PROGRAM-ID. GETPRML.
                    AUTHOR. EXAMPLE.
                    DATE-WRITTEN.   03/25/98.

                    ENVIRONMENT DIVISION.
                    INPUT-OUTPUT SECTION.
                    FILE-CONTROL.
                    DATA DIVISION.
                    FILE SECTION.

                   WORKING-STORAGE SECTION.

                        EXEC SQL INCLUDE SQLCA END-EXEC.
              **************************************************
              *   DECLARE A HOST VARIABLE TO HOLD INPUT SCHEMA
              **************************************************
               01  INSCHEMA PIC X(8).


              **************************************************
              *   DECLARE CURSOR FOR RETURNING RESULT SETS
              **************************************************
              *
                   EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
                     SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
                   END-EXEC.
              *
               LINKAGE SECTION.
              **************************************************
              *   DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
              **************************************************
               01  PROCNM  PIC X(18).
               01  SCHEMA  PIC X(8).
              ****************************************************
              *   DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
              ****************************************************
               01  OUT-CODE PIC S9(9) USAGE BINARY.
               01  PARMLST.
                   49 PARMLST-LEN  PIC S9(4) USAGE BINARY.
                   49 PARMLST-TEXT PIC X(254).

               PROCEDURE DIVISION USING PROCNM, SCHEMA,
                       OUT-CODE, PARMLST.
```

Figure 253. A COBOL stored procedure with linkage convention GENERAL (Part 1 of 2)

```
*******************************************************
* Issue the SQL SELECT against the SYSIBM.SYSROUTINES
* DB2 catalog table.
*******************************************************
      EXEC SQL
        SELECT RUNOPTS INTO :PARMLST
          FROM SYSIBM.ROUTINES
          WHERE NAME=:PROCNM AND
          SCHEMA=:SCHEMA
      END-EXEC.

*******************************************************
*  COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
*******************************************************
       MOVE SQLCODE TO OUT-CODE.
*******************************************************
* OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
* TO THE CALLER.
*******************************************************
      EXEC SQL OPEN C1
      END-EXEC.
 PROG-END.
       GOBACK.
```

Figure 253. A COBOL stored procedure with linkage convention GENERAL (Part 2 of 2)

COBOL stored procedure: GENERAL WITH NULLS

This example stored procedure does the following:

- Searches the DB2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.

- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE COBOL
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL WITH NULLS
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT SAMPPROG
  PROGRAM TYPE MAIN
  SECURITY DB2
  RESULT SETS 2
  COMMIT ON RETURN NO;
```

```
          CBL RENT
          IDENTIFICATION DIVISION.
          PROGRAM-ID. GETPRML.
          AUTHOR. EXAMPLE.
          DATE-WRITTEN.   03/25/98.

          ENVIRONMENT DIVISION.
          INPUT-OUTPUT SECTION.
          FILE-CONTROL.
          DATA DIVISION.
          FILE SECTION.
*
          WORKING-STORAGE SECTION.
*
              EXEC SQL INCLUDE SQLCA END-EXEC.
*
          **************************************************
*         DECLARE A HOST VARIABLE TO HOLD INPUT SCHEMA
          **************************************************
          01  INSCHEMA PIC X(8).
          **************************************************
*         DECLARE CURSOR FOR RETURNING RESULT SETS
          **************************************************
*
              EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
                SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
              END-EXEC.
*
          LINKAGE SECTION.
          **************************************************
*         DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
          **************************************************
          01  PROCNM  PIC X(18).
          01  SCHEMA  PIC X(8).
          **************************************************
*         DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
          **************************************************
          01  OUT-CODE PIC S9(9) USAGE BINARY.
          01  PARMLST.
              49 PARMLST-LEN  PIC S9(4) USAGE BINARY.
              49 PARMLST-TEXT PIC X(254).
          **************************************************
*         DECLARE THE STRUCTURE CONTAINING THE NULL
*         INDICATORS FOR THE INPUT AND OUTPUT PARAMETERS.
          **************************************************
          01  IND-PARM.
              03 PROCNM-IND   PIC S9(4) USAGE BINARY.
              03 SCHEMA-IND   PIC S9(4) USAGE BINARY.
              03 OUT-CODE-IND PIC S9(4) USAGE BINARY.
              03 PARMLST-IND  PIC S9(4) USAGE BINARY.
```

*Figure 254. A COBOL stored procedure with linkage convention GENERAL WITH NULLS
(Part 1 of 2)*

```
                 PROCEDURE DIVISION USING PROCNM, SCHEMA,
                        OUT-CODE, PARMLST, IND-PARM.
       ********************************************************
       * If any input parameter is null, return a null value
       * for PARMLST and set the output return code to 9999.
       ********************************************************
            IF PROCNM-IND < 0 OR
               SCHEMA-IND < 0
                  MOVE 9999 TO OUT-CODE
                  MOVE 0 TO OUT-CODE-IND
                  MOVE -1 TO PARMLST-IND
            ELSE
       ********************************************************
       * Issue the SQL SELECT against the SYSIBM.SYSROUTINES
       * DB2 catalog table.
       ********************************************************
            EXEC SQL
              SELECT RUNOPTS INTO :PARMLST
              FROM SYSIBM.SYSROUTINES
              WHERE NAME=:PROCNM AND
              SCHEMA=:SCHEMA
            END-EXEC
                  MOVE 0 TO PARMLST-IND
       ********************************************************
       *  COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
       ********************************************************
                  MOVE SQLCODE TO OUT-CODE
                  MOVE 0 TO OUT-CODE-IND.
       *
       ********************************************************
       * OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
       * TO THE CALLER.
       ********************************************************
            EXEC SQL OPEN C1
            END-EXEC.
        PROG-END.
            GOBACK.
```

*Figure 254. A COBOL stored procedure with linkage convention GENERAL WITH NULLS
(Part 2 of 2)*

PL/I stored procedure: GENERAL

This example stored procedure searches the DB2 SYSIBM.SYSROUTINES catalog
table for a row that matches the input parameters from the client program. The two
input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the
SELECT operation, and the value of the RUNOPTS column retrieved from the
SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE PLI
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
```

```
   ASUTIME NO LIMIT
   PARAMETER STYLE GENERAL
   STAY RESIDENT NO
   RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
   WLM ENVIRONMENT SAMPPROG
   PROGRAM TYPE MAIN
   SECURITY DB2
   RESULT SETS 0
   COMMIT ON RETURN NO;


*PROCESS SYSTEM(MVS);

 GETPRML:
   PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST)
                     OPTIONS(MAIN NOEXECOPS REENTRANT);

   DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
           SCHEMA CHAR(8),     /* INPUT parm -- User's SCHEMA  */

           OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from   */
                                   /*   the SELECT operation.  */
           PARMLST CHAR(254)      /* OUTPUT -- RUNOPTS for     */
                   VARYING;       /*   the matching row in     */
                                  /*   SYSIBM.SYSROUTINES       */

   EXEC SQL INCLUDE SQLCA;

   /*************************************************************/
   /* Execute SELECT from SYSIBM.SYSROUTINES in the catalog.  */
   /*************************************************************/
   EXEC SQL
     SELECT RUNOPTS INTO :PARMLST
         FROM SYSIBM.SYSROUTINES
         WHERE NAME=:PROCNM AND
               SCHEMA=:SCHEMA;

   OUT_CODE = SQLCODE;          /* return SQLCODE to caller    */
   RETURN;
 END GETPRML;
```

Figure 255. A PL/I stored procedure with linkage convention GENERAL

PL/I stored procedure: GENERAL WITH NULLS

This example stored procedure searches the DB2 SYSIBM.SYSROUTINES catalog
table for a row that matches the input parameters from the client program. The two
input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the
SELECT operation, and the value of the RUNOPTS column retrieved from the
SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE PLI
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME 'GETPRML'
  COLLID GETPRML
```

```
           ASUTIME NO LIMIT
           PARAMETER STYLE GENERAL WITH NULLS
           STAY RESIDENT NO
           RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
           WLM ENVIRONMENT SAMPPROG
           PROGRAM TYPE MAIN
           SECURITY DB2
           RESULT SETS 0
           COMMIT ON RETURN NO;


     *PROCESS SYSTEM(MVS);

      GETPRML:
        PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST, INDICATORS)
                         OPTIONS(MAIN NOEXECOPS REENTRANT);

        DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
                SCHEMA CHAR(8),     /* INPUT parm -- User's schema  */

                OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from   */
                                    /*    the SELECT operation. */
                PARMLST CHAR(254)       /* OUTPUT -- PARMLIST for   */
                      VARYING;          /*   the matching row in    */
                                        /*   SYSIBM.SYSROUTINES     */
        DECLARE 1 INDICATORS,       /* Declare null indicators for  */
                                    /* input and output parameters. */
                3 PROCNM_IND   FIXED BIN(15),
                3 SCHEMA_IND   FIXED BIN(15),
                3 OUT_CODE_IND FIXED BIN(15),
                3 PARMLST_IND  FIXED BIN(15);

        EXEC SQL INCLUDE SQLCA;

        IF PROCNM_IND<0 |
           SCHEMA_IND<0 THEN
         DO;                    /* If any input parm is NULL,    */
           OUT_CODE = 9999;     /* Set output return code.       */
           OUT_CODE_IND = 0;
                                /* Output return code is not NULL.*/
           PARMLST_IND = -1;    /* Assign NULL value to PARMLST.  */
         END;
        ELSE                    /* If input parms are not NULL,   */
         DO;                    /*                                */
        /************************************************************/
        /* Issue the SQL SELECT against the SYSIBM.SYSROUTINES      */
        /* DB2 catalog table.                                       */
        /************************************************************/
           EXEC SQL
             SELECT RUNOPTS INTO :PARMLST
                 FROM SYSIBM.SYSROUTINES
                 WHERE NAME=:PROCNM AND
                      SCHEMA=:SCHEMA;
           PARMLST_IND = 0;     /* Mark PARMLST as not NULL.      */

           OUT_CODE = SQLCODE;  /* return SQLCODE to caller       */
           OUT_CODE_IND = 0;
           OUT_CODE_IND = 0;    /* Output return code is not NULL.*/
         END;
        RETURN;

      END GETPRML;
```

Figure 256. A PL/I stored procedure with linkage convention GENERAL WITH NULLS

Appendix E. REBIND subcommands for lists of plans or packages

If a list of packages or plans that you want to rebind is not easily specified using asterisks, you might be able to create the needed REBIND subcommands automatically, using the sample program DSNTIAUL.

One situation in which this technique might be useful is when a resource becomes unavailable during a rebind of many plans or packages. DB2 normally terminates the rebind and does not rebind the remaining plans or packages. Later, however, you might want to rebind only the objects that remain to be rebound. You can build REBIND subcommands for the remaining plans or packages by using DSNTIAUL to select the plans or packages from the DB2 catalog and to create the REBIND subcommands. You can then submit the subcommands through the DSN command processor, as usual.

You might first need to edit the output from DSNTIAUL so that DSN can accept it as input. The CLIST DSNTEDIT can perform much of that task for you.

This section contains the following topics:
- Overview of the procedure for generating lists of REBIND commands
- "Sample SELECT statements for generating REBIND commands"
- "Sample JCL for running lists of REBIND commands" on page 914

Overview of the procedure for generating lists of REBIND commands

Figure 257 shows an overview of the procedures for REBIND PLAN and REBIND PACKAGE.

Procedure for REBIND PLAN

| Use DSNTIAUL to generate the REBIND PLAN subcommands for the selected plans. |

↓

| Use TSO edit commands to add TSO DSN commands to the sequential data set. |

↓

| Use DSN to execute the REBIND PLAN subcommands for the selected plans. |

Procedure for REBIND PACKAGE

| Use DSNTIAUL to generate the REBIND PACKAGE subcommands for the selected packages. |

↓

| Use DSNTEDIT CLIST to delete extraneous blanks from the REBIND PACKAGE subcommands. |

↓

| Use TSO edit commands to add DSN commands to the sequential data set. |

↓

| Use DSN to execute the REBIND PACKAGE subcommands for the selected packages. |

Figure 257. Procedures for executing lists of REBIND commands

Sample SELECT statements for generating REBIND commands

Building REBIND subcommands: The examples that follow illustrate the following techniques:
- Using SELECT to select specific packages or plans to be rebound

- Using the CONCAT operator to concatenate the REBIND subcommand syntax around the plan or package names
- Using the SUBSTR function to convert a varying-length string to a fixed-length string
- Appending additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input

If the SELECT statement returns rows, then DSNTIAUL generates REBIND subcommands for the plans or packages identified in the returned rows. Put those subcommands in a sequential dataset, where you can then edit them.

For REBIND PACKAGE subcommands, delete any extraneous blanks in the package name, using either TSO edit commands or the DB2 CLIST DSNTEDIT.

For both REBIND PLAN and REBIND PACKAGE subcommands, add the DSN command that the statement needs as the first line in the sequential dataset, and add END as the last line, using TSO edit commands. When you have edited the sequential dataset, you can run it to rebind the selected plans or packages.

If the SELECT statement returns no qualifying rows, then DSNTIAUL does not generate REBIND subcommands.

The examples in this section generate REBIND subcommands that work in DB2 for OS/390 and z/OS Version 7. You might need to modify the examples for prior releases of DB2 that do not allow all of the same syntax.

Example 1:
REBIND all plans without terminating because of unavailable resources.
```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                    ',1,45)
FROM SYSIBM.SYSPLAN;
```

Example 2:
REBIND all versions of all packages without terminating because of unavailable resources.
```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))       ',1,55)
FROM SYSIBM.SYSPACKAGE;
```

Example 3:
REBIND all plans bound before a given date and time.
```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                    ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE <= 'yyyymmdd' AND
  BINDTIME <= 'hhmmssth';
```

where *yyyymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 4:
REBIND all versions of all packages bound before a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME <= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example 5:
REBIND all plans bound since a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                      ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE >= 'yyyymmdd' AND
  BINDTIME >= 'hhmmssth';
```

where *yyyymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 6:
REBIND all versions of all packages bound since a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID
  CONCAT'.'CONCAT NAME
  CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example 7:
REBIND all plans bound within a given date and time range.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                      ',1,45)
FROM SYSIBM.SYSPLAN
WHERE
  (BINDDATE >= 'yyyymmdd' AND
   BINDTIME >= 'hhmmssth') AND
   BINDDATE <= 'yyyymmdd' AND
   BINDTIME <= 'hhmmssth');
```

where *yyyymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 8:
REBIND all versions of all packages bound within a given date and time range.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp1' AND
  BINDTIME <= 'timestamp2';
```

where *timestamp1* and *timestamp2* are ISO timestamp strings.

Example 9:
> REBIND all invalid plans.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                    ',1,45)
FROM SYSIBM.SYSPLAN
WHERE VALID = 'N';
```

Example 10:
> REBIND all invalid versions of all packages.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))      ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE VALID = 'N';
```

Example 11:
> REBIND all plans bound with ISOLATION level of cursor stability.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT')                    ',1,45)
FROM SYSIBM.SYSPLAN
WHERE ISOLATION = 'S';
```

Example 12:
> REBIND all versions of all packages that allow CPU and/or I/O parallelism.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))      ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE DEGREE='ANY';
```

Sample JCL for running lists of REBIND commands

Figure 258 shows the JCL to rebind all versions of all packages bound in 1994.

Figure 259 on page 916 shows some sample JCL for rebinding all plans bound without specifying the DEGREE keyword on BIND with DEGREE(ANY).

```
//REBINDS  JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//             REGION=1024K
//********************************************************************/
//SETUP    EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB71) PARMS('SQL') -
      LIB('DSN710.RUNLIB.LOAD')
 END
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//             UNIT=SYSDA,DISP=SHR
```

Figure 258. Example JCL: Rebind all packages bound in 1994. (Part 1 of 2)

```
//********************************************************************/
//*
//* GENER= '<SUBCOMMANDS TO REBIND ALL PACKAGES BOUND IN 1994
//*
//********************************************************************/
//SYSIN    DD *
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'
  CONCAT NAME CONCAT'.(*))          ',1,55)
   FROM SYSIBM.SYSPACKAGE
   WHERE BINDTIME >= '1994-01-01-00.00.00.000000' AND
         BINDTIME <= '1994-12-31-23.59.59.999999';
/*
//********************************************************************/
//*
//* STRIP THE BLANKS OUT OF THE REBIND SUBCOMMANDS
//*
//********************************************************************/
//STRIP    EXEC PGM=IKJEFT01
//SYSPROC  DD DSN=SYSADM.DSNCLIST,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
 DSNTEDIT SYSADM.SYSTSIN.DATA
//SYSIN    DD DUMMY
/*
//********************************************************************/
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//********************************************************************/
//EDIT     EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
 TOP
 INSERT DSN SYSTEM(DSN)
 BOTTOM
 INSERT END
 TOP
 LIST * 99999
 END SAVE
/*
//********************************************************************/
//*
//* EXECUTE THE REBIND PACKAGE SUBCOMMANDS THROUGH DSN
//*
//********************************************************************/
//LOCAL    EXEC PGM=IKJEFT01
//DBRMLIB  DD DSN=DSN710.DBRMLIB.DATA,
//            DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD DSN=SYSADM.SYSTSIN.DATA,
//            UNIT=SYSDA,DISP=SHR
/*
```

Figure 258. Example JCL: Rebind all packages bound in 1994. (Part 2 of 2)

```
//REBINDS  JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//             REGION=1024K
//*******************************************************************/
//SETUP    EXEC TSOBATCH
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//             UNIT=SYSDA,DISP=SHR
//*******************************************************************/
//*
//* REBIND ALL PLANS THAT WERE BOUND WITHOUT SPECIFYING THE DEGREE
//* KEYWORD ON BIND WITH DEGREE(ANY)
//*
//*******************************************************************/
//SYSTSIN DD *
 DSN S(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB71) PARM('SQL')
 END
//SYSIN DD *
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
  CONCAT') DEGREE(ANY)             ',1,45)
   FROM SYSIBM.SYSPLAN
   WHERE DEGREE = ' ';
/*
//*******************************************************************/
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//*******************************************************************/
//EDIT    EXEC PGM=IKJEFT01
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
 TOP
 INSERT DSN S(DSN)
 BOTTOM
 INSERT END
 TOP
 LIST * 99999
 END SAVE
/*
//*******************************************************************/
//*
//* EXECUTE THE REBIND SUBCOMMANDS THROUGH DSN
//*
//*******************************************************************/
//REBIND   EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=SYSADM.TESTLIB,DISP=SHR
//             DD DSN=DSN710.SDSNLOAD,DISP=SHR
//DBRMLIB  DD DSN=SYSADM.DBRMLIB.DATA,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD DSN=SYSADM.SYSTSIN.DATA,DISP=SHR
//SYSIN    DD DUMMY
/*
```

Figure 259. Example JCL: Rebind selected plans with a different bind option

Appendix F. SQL reserved words

Table 130 on page 918 lists the words that cannot be used as ordinary identifiers in some contexts because they might be interpreted as SQL keywords. For example, ALL cannot be a column name in a SELECT statement. Each word, however, can be used as a delimited identifier in contexts where it otherwise cannot be used as an ordinary identifier. For example, if the quotation mark (") is the escape character that begins and ends delimited identifiers, "ALL" can appear as a column name in a SELECT statement. In addition, some sections of this book might indicate words that cannot be used in the specific context that is being described.

Table 130. SQL reserved words

ADD	CURRENT_TIMESTAMP	GOTO	NULL	SECQTY
AFTER	CURSOR	GRANT	NULLS	SECURITY
ALL	DATA	GROUP	NUMPARTS	SELECT
ALLOCATE	DATABASE	HANDLER	OBID	SENSITIVE
ALLOW	DAY	HAVING	OF	SET
ALTER	DAYS	HOUR	ON	SIMPLE
AND	DBINFO	HOURS	OPEN	SOME
ANY	DB2SQL	IF	OPTIMIZATION	SOURCE
APPLICATION	DECLARE	IMMEDIATE	OPTIMIZE	SPECIFIC
AS	DEFAULT	IN	OR	STANDARD
ASSOCIATE	DELETE	INDEX	ORDER	STATIC
ASUTIME	DESCRIPTOR	INHERIT	OUT	STAY
AUDIT	DETERMINISTIC	INNER	OUTER	STOGROUP
AUX	DISALLOW	INOUT	PACKAGE	STORES
AUXILIARY	DISTINCT	INSENSITIVE	PARAMETER	STYLE
BEFORE	DO	INSERT	PART	SUBPAGES
BEGIN	DOUBLE	INTO	PATH	SYNONYM
BETWEEN	DROP	IS	PIECESIZE	SYSFUN
BUFFERPOOL	DSNHATTR	ISOBID	PLAN	SYSIBM
BY	DSSIZE	JAR	PRECISION	SYSPROC
CALL	DYNAMIC	JAVA	PREPARE	SYSTEM
CAPTURE	EDITPROC	JOIN	PRIQTY	TABLE
CASCADED	ELSE	KEY	PRIVILEGES	TABLESPACE
CASE	ELSEIF	LABEL	PROCEDURE	THEN
CAST	ENCODING	LANGUAGE	PROGRAM	TO
CCSID	END	LC_CTYPE	PSID	TRIGGER
CHAR	END-EXEC[1]	LEAVE	QUERYNO	UNDO
CHARACTER	ERASE	LEFT	READS	UNION
CHECK	ESCAPE	LIKE	REFERENCES	UNIQUE
CLOSE	EXCEPT	LOCAL	RELEASE	UNTIL
CLUSTER	EXECUTE	LOCALE	RENAME	UPDATE
COLLECTION	EXISTS	LOCATOR	REPEAT	USER
COLLID	EXIT	LOCATORS	RESTRICT	USING
COLUMN	EXTERNAL	LOCK	RESULT	VALIDPROC
COMMENT	FENCED	LOCKMAX	RESULT_SET_LOCATOR	VALUES
COMMIT	FETCH	LOCKSIZE	RETURN	VARIANT
CONCAT	FIELDPROC	LONG	RETURNS	VCAT
CONDITION	FINAL	LOOP	REVOKE	VIEW
CONNECT	FOR	MICROSECOND	RIGHT	VOLUMES
CONNECTION	FROM	MICROSECONDS	ROLLBACK	WHEN
CONSTRAINT	FULL	MINUTE	RUN	WHERE
CONTAINS	FUNCTION	MINUTES	SAVEPOINT	WHILE
CONTINUE	GENERAL	MODIFIES	SCHEMA	WITH
CREATE	GENERATED	MONTH	SCRATCHPAD	WLM
CURRENT	GET	MONTHS	SECOND	YEAR
CURRENT_DATE	GLOBAL	NO	SECONDS	YEARS
CURRENT_LC_CTYPE	GO	NOT		
CURRENT_PATH				
CURRENT_TIME				

Note: [1] COBOL only

IBM SQL has additional reserved words that DB2 for OS/390 and z/OS does not enforce. Therefore, we suggest that you do not use these additional reserved words as ordinary identifiers in names that have a continuing use. See *IBM SQL Reference* for a list of the words.

Appendix G. Characteristics of SQL statements in DB2 for OS/390 and z/OS

This appendix provides a summary of the actions that are allowed on SQL statements in DB2 for OS/390 and z/OS. It also contains a list of the SQL statements that can be executed in external user-defined functions and stored procedures and in SQL procedures.

Actions allowed on SQL statements

Table 131 shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 131. Actions allowed on SQL statements in DB2 for OS/390 and z/OS

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALLOCATE CURSOR[1]	Y	Y	Y		
ALTER[2]	Y	Y		Y	
ASSOCIATE LOCATORS[1]	Y	Y	Y		
BEGIN DECLARE SECTION					Y
CALL[1]	Y			Y	
CLOSE	Y			Y	
COMMENT ON	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2)	Y		Y		
CREATE[2]	Y	Y		Y	
DECLARE CURSOR					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DECLARE STATEMENT					Y
DECLARE TABLE					Y
DELETE	Y	Y		Y	
DESCRIBE prepared statement or table	Y			Y	
DESCRIBE CURSOR	Y		Y		
DESCRIBE INPUT	Y			Y	
DESCRIBE PROCEDURE	Y		Y		
DROP[2]	Y	Y		Y	
END DECLARE SECTION					Y
EXECUTE	Y			Y	
EXECUTE IMMEDIATE	Y			Y	
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	

SQL statement	Executable	Interactively or dynamically prepared	Processed by Requesting system	Server	Precompiler
FREE LOCATOR[1]	Y	Y		Y	
GRANT[2]	Y	Y		Y	
HOLD LOCATOR[1]	Y	Y		Y	
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL ON	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y[4]	
RELEASE connection	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME[2]	Y	Y		Y	
REVOKE[2]	Y	Y		Y	
ROLLBACK	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT APPLICATION ENCODING SCHEME	Y		Y		
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT LC_CTYPE	Y	Y		Y	
SET CURRENT OPTIMIZATION HINT	Y	Y		Y	
SET CURRENT PACKAGESET	Y		Y		
SET CURRENT PATH	Y	Y		Y	
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT RULES	Y	Y		Y	
SET CURRENT SQLID[5]	Y	Y		Y	
SET host-variable = CURRENT APPLICATION ENCODING SCHEME	Y	Y	Y		
SET host-variable = CURRENT DATE	Y			Y	
SET host-variable = CURRENT DEGREE	Y			Y	
SET host-variable = CURRENT MEMBER	Y			Y	
SET host-variable = CURRENT PACKAGESET	Y		Y		
SET host-variable = CURRENT PATH	Y			Y	

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
SET *host-variable* = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET *host-variable* = CURRENT SERVER	Y		Y		
SET *host-variable* = CURRENT SQLID	Y			Y	
SET *host-variable* = CURRENT TIME	Y			Y	
SET *host-variable* = CURRENT TIMESTAMP	Y			Y	
SET *host-variable* = CURRENT TIMEZONE	Y			Y	
SET *transition-variable* = CURRENT DATE	Y			Y	
SET *transition-variable* = CURRENT DEGREE	Y			Y	
SET *transition-variable* = CURRENT PATH	Y			Y	
SET *transition-variable* = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET *transition-variable* = CURRENT SQLID	Y			Y	
SET *transition-variable* = CURRENT TIME	Y			Y	
SET *transition-variable* = CURRENT TIMESTAMP	Y			Y	
SET *transition-variable* = CURRENT TIMEZONE	Y			Y	
SIGNAL SQLSTATE[6]	Y			Y	
UPDATE	Y	Y		Y	
VALUES[6]	Y			Y	
VALUES INTO[7]	Y			Y	
WHENEVER					Y

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler

Notes:

1. The statement can be dynamically prepared. It cannot be issued dynamically.

2. The statement can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

3. The statement can be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements.

4. The requesting system processes the PREPARE statement when the statement being prepared is ALLOCATE CURSOR or ASSOCIATE LOCATORS.

5. The value to which special register CURRENT SQLID is set is used as the SQL authorization ID and the implicit qualifier for dynamic SQL statements only when DYNAMICRULES run behavior is in effect. The CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.

6. This statement can only be used in the triggered action of a trigger.

7. Local special registers can be referenced in a VALUES INTO statement if it results in the assignment of a single host-variable, not if it results in setting more than one value.

SQL statements allowed in external functions and stored procedures

Table 132 shows which SQL statements in an external stored procedure or in an external user-defined function can execute. Whether the statements can be executed depends on the level of SQL data access with which the stored procedure or external function is defined (NO SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA). The letter **Y** means *yes*.

In general, if an executable SQL statement is encountered in a stored procedure or function defined as NO SQL, SQLSTATE 38001 is returned. If the routine is defined to allow some level of SQL access, SQL statements that are not supported in any context return SQLSTATE 38003. SQL statements not allowed for routines defined as CONTAINS SQL return SQLSTATE 38004, and SQL statements not allowed for READS SQL DATA return SQL STATE 38002.

Table 132. SQL statements in external user-defined functions and stored procedures

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALLOCATE CURSOR			Y	Y
ALTER				Y
ASSOCIATE LOCATORS			Y	Y
BEGIN DECLARE SECTION	Y[1]	Y	Y	Y
CALL		Y[2]	Y[2]	Y[2]
CLOSE			Y	Y
COMMENT ON				Y
COMMIT[3]		Y	Y	Y
CONNECT (Type 1 and Type 2)		Y	Y	Y
CREATE				Y

Table 132. SQL statements in external user-defined functions and stored procedures (continued)

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
DECLARE CURSOR	Y[1]	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DECLARE STATEMENT	Y[1]	Y	Y	Y
DECLARE TABLE	Y[1]	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DESCRIBE CURSOR			Y	Y
DESCRIBE INPUT			Y	Y
DESCRIBE PROCEDURE			Y	Y
DROP				Y
END DECLARE SECTION	Y[1]	Y	Y	Y
EXECUTE		Y[4]	Y[4]	Y
EXECUTE IMMEDIATE		Y[4]	Y[4]	Y
EXPLAIN				Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GRANT				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y[1]	Y	Y	Y
INSERT				Y
LABEL ON				Y
LOCK TABLE		Y	Y	Y
OPEN			Y	Y
PREPARE		Y	Y	Y
RELEASE connection		Y	Y	Y
RELEASE SAVEPOINT[6]				Y
REVOKE				Y
ROLLBACK[6, 7, 8]		Y	Y	Y
ROLLBACK TO SAVEPOINT[6, 7, 8]				Y
SAVEPOINT[6]				Y
SELECT			Y	Y
SELECT INTO			Y	Y
SET CONNECTION		Y	Y	Y
SET host-variable Assignment		Y[5]	Y	Y
SET special register		Y	Y	Y

	Level of SQL access			
SQL statement	**NO SQL**	**CONTAINS SQL**	**READS SQL DATA**	**MODIFIES SQL DATA**
SET transition-variable Assignment		Y[5]	Y	Y
SIGNAL SQLSTATE		Y	Y	Y
UPDATE				Y
VALUES			Y	Y
VALUES INTO		Y[5]	Y	Y
WHENEVER	Y[1]	Y	Y	Y

Notes:

1. Although the SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.

2. The stored procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a stored procedure defined as MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL. A routine defined as CONTAINS SQL can only call a procedure defined as CONTAINS SQL.

3. The COMMIT statement cannot be executed in a user-defined function. The COMMIT statement cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.

4. The statement specified for the EXECUTE statment must be a statement that is allowed for the particular level of SQL data access in effect. For example, if the level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, or DELETE.

5. The statement is supported only if it does not contain a subquery or query-expression.

6. RELEASE SAVEPOINT, SAVEPOINT, and ROLLBACK (with the TO SAVEPOINT clause) cannot be executed from a user-defined function.

7. If the ROLLBACK statement (without the TO SAVEPOINT clause) is executed in a user-defined function, an error is returned to the calling program, and the application is placed in a *must rollback* state.

8. The ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.

SQL statements allowed in SQL procedures

Table 133 on page 925 lists the statements that are valid in an SQL procedure body, in addition to SQL procedure statements. The table lists the statements that can be used as the only statement in the SQL procedure and as the statements that can be nested in a compound statement. An SQL statement can be executed in an SQL procedure depending on whether MODIFIES SQL DATA, CONTAINS SQL, or READS SQL DATA is specified in the stored procedure definition. See Table 132 on page 922 for a list of SQL statements that can be executed for each of these parameter values.

Table 133. Valid SQL statements in an SQL procedure body

| | SQL statement is... | |
| | The only statement in the procedure | Nested in a compound statement |
SQL statement		
ALLOCATE CURSOR		Y
ALTER DATABASE	Y	Y
ALTER FUNCTION	Y	Y
ALTER INDEX	Y	Y
ALTER PROCEDURE	Y	Y
ALTER STOGROUP	Y	Y
ALTER TABLE	Y	Y
ALTER TABLESPACE	Y	Y
ASSOCIATE LOCATORS		Y
BEGIN DECLARE SECTION		
CALL		Y
CLOSE		Y
COMMENT ON	Y	Y
COMMIT[1]	Y	Y
CONNECT (Type 1 and Type 2)	Y	Y
CREATE ALIAS	Y	Y
CREATE DATABASE	Y	Y
CREATE DISTINCT TYPE	Y	Y
CREATE FUNCTION[2]	Y	Y
CREATE GLOBAL TEMPORARY TABLE	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE[2]	Y	Y
CREATE STOGROUP	Y	Y
CREATE SYNONYM	Y	Y
CREATE TABLE	Y	Y
CREATE TABLESPACE	Y	Y
CREATE TRIGGER		
CREATE VIEW	Y	Y
DECLARE CURSOR		Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DECLARE STATEMENT		
DECLARE TABLE		
DELETE	Y	Y
DESCRIBE prepared statement or table		
DESCRIBE CURSOR		
DESCRIBE INPUT		
DESCRIBE PROCEDURE		

Table 133. Valid SQL statements in an SQL procedure body (continued)

	SQL statement is...	
SQL statement	**The only statement in the procedure**	**Nested in a compound statement**
DROP	Y	Y
END DECLARE SECTION		
EXECUTE		Y
EXECUTE IMMEDIATE	Y	Y
EXPLAIN		
FETCH		Y
FREE LOCATOR		
GRANT	Y	Y
HOLD LOCATOR		
INCLUDE		
INSERT	Y	Y
LABEL ON	Y	Y
LOCK TABLE	Y	Y
OPEN		Y
PREPARE FROM		Y
RELEASE connection	Y	Y
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK[1]	Y	Y
ROLLBACK TO SAVEPOINT[1]	Y	Y
SAVEPOINT	Y	Y
SELECT		
SELECT INTO	Y	Y
SET CONNECTION	Y	Y
SET host-variable Assignment[3]		
SET special register[3]	Y	Y
SET transition-variable Assignment[3]		
SIGNAL SQLSTATE		
UPDATE	Y	Y
VALUES		
VALUES INTO	Y	Y
WHENEVER		

Table 133. Valid SQL statements in an SQL procedure body (continued)

	SQL statement is...	
SQL statement	**The only statement in the procedure**	**Nested in a compound statement**

Notes:

1. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger

2. CREATE FUNCTION with LANGUAGE SQL (specified either implicitly or explicitly) and CREATE PROCEDURE with LANGUAGE SQL are not allowed within the body of an SQL procedure.

3. SET host-variable assignment, SET transition-variable assignment, and SET special register are the SQL SET statements, not the SQL procedure assignment statement.

Appendix H. Program preparation options for remote packages

The table that follows gives generic descriptions of program preparation options, lists the equivalent DB2 option for each one, and indicates if appropriate, whether it is a bind package (B) or a precompiler (P) option. In addition, the table indicates whether a DB2 server supports the option.

Table 134. Program preparation options for packages

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package replacement: protect existing packages	ACTION(ADD)	B	Supported
Package replacement: replace existing packages	ACTION(REPLACE)	B	Supported
Package replacement: version name	ACTION(REPLACE REPLVER (version-id))	B	Supported
Statement string delimiter	APOSTSQL/QUOTESQL	P	Supported
DRDA access: SQL CONNECT (Type 1)	CONNECT(1)	P	Supported
DRDA access: SQL CONNECT (Type 2)	CONNECT(2)	P	Supported
Block protocol: Do not block data for an ambiguous cursor	CURRENTDATA(YES)	B	Supported
Block protocol: Block data when possible	CURRENTDATA(NO)	B	Supported
Block protocol: Never block data	(Not available)		Not supported
Name of remote database	CURRENTSERVER(location name)	B	Supported as a BIND PLAN option
Date format of statement	DATE	P	Supported
Protocol for remote access	DBPROTOCOL	B	Not supported
Maximum decimal precision: 15	DEC(15)	P	Supported
Maximum decimal precision: 31	DEC(31)	P	Supported
Defer preparation of dynamic SQL	DEFER(PREPARE)	B	Supported
Do not defer preparation of dynamic SQL	NODEFER(PREPARE)	B	Supported
Dynamic SQL Authorization	DYNAMICRULES	B	Supported
Encoding scheme for static SQL statements	ENCODING	B	Not supported
Explain option	EXPLAIN	B	Supported
Immediately write group bufferpool-dependent page sets or partitions in a data sharing environment	IMMEDWRITE	B	Supported
Package isolation level: CS	ISOLATION(CS)	B	Supported
Package isolation level: RR	ISOLATION(RR)	B	Supported
Package isolation level: RS	ISOLATION(RS)	B	Supported

Table 134. Program preparation options for packages (continued)

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package isolation level: UR	ISOLATION(UR)	B	Supported
Keep prepared statements after commit points	KEEPDYNAMIC	B	Supported
Consistency token	LEVEL	P	Supported
Package name	MEMBER	B	Supported
Package owner	OWNER	B	Supported
Schema name list for user-defined functions, distinct types, and stored procedures	PATH	B	Supported
Statement decimal delimiter	PERIOD/COMMA	P	Supported
Default qualifier	QUALIFIER	B	Supported
Use access path hints	OPTHINT	B	Supported
Lock release option	RELEASE	B	Supported
Choose access path at run time	REOPT(VARS)	B	Supported
Choose access path at bind time only	NOREOPT(VARS)	B	Supported
Creation control: create a package despite errors	SQLERROR(CONTINUE)	B	Supported
Creation control: create no package if there are errors	SQLERROR(NO PACKAGE)	B	Supported
Creation control: create no package	(Not available)		Supported
Time format of statement	TIME	P	Supported
Existence checking: full	VALIDATE(BIND)	B	Supported
Existence checking: deferred	VALIDATE(RUN)	B	Supported
Package version	VERSION	P	Supported
Default character subtype: system default	(Not available)		Supported
Default character subtype: BIT	(Not available)		Not supported
Default character subtype: SBCS	(Not available)		Not supported
Default character subtype: DBCS	(Not available)		Not supported
Default character CCSID: SBCS	(Not available)		Not supported
Default character CCSID: Mixed	(Not available)		Not supported
Default character CCSID: Graphic	(Not available)		Not supported
Package label	(Not available)		Ignored when received; no error is returned
Privilege inheritance: retain	default		Supported
Privilege inheritance: revoke	(Not available)		Not supported

Appendix I. Stored procedures shipped with DB2

DB2 provides several stored procedures that you can call in your application programs to perform a number of utility and application programming functions. Those stored procedures are:

- The utilities stored procedure (DSNUTILS)

  This stored procedure lets you invoke utilities from a local or remote client program. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The DB2 UDB Control Center table space and index information stored procedure (DSNACCQC)

  This stored procedure helps you determine when utilities should be run on your databases. This stored procedure is designed primarily for use by the DB2 UDB Control Center but can be invoked from any client program. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The DB2 UDB Control Center partition information stored procedure (DSNACCAV)

  This stored procedure helps you determine when utilities should be run on your partitioned table spaces. This stored procedure is designed primarily for use by the DB2 UDB Control Center but can be invoked from any client program. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The WLM environment refresh stored procedure (WLM_REFRESH)

  This stored procedure lets you refresh a WLM environment from a remote workstation. See "The WLM environment refresh stored procedure (WLM_REFRESH)" for information.

The WLM environment refresh stored procedure (WLM_REFRESH)

The WLM_REFRESH stored procedure refreshes a WLM environment. WLM_REFRESH can recycle the environment in which it runs, as well as any other WLM environment.

Environment

WLM_REFRESH runs in a WLM-established stored procedures address space. The load module for WLM_REFRESH, DSNTWR, must reside in an APF-authorized library.

Authorization required

To execute the CALL statement, the SQL authorization ID of the process must have READ access or higher to the OS/390 Security Server System Authorization Facility (SAF) resource profile *ssid*.WLM_REFRESH.*WLM-environment-name* in resource class DSNR. See Part 3 (Volume 1) of *DB2 Administration Guide* for information on authorizing access to SAF resource profiles.

WLM_REFRESH syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking WLM_REFRESH. Because the linkage convention for WLM_REFRESH is GENERAL, you cannot pass null values to the stored procedure.

```
►►──CALL──WLM_REFRESH──(──WLM-environment,──ssid,──status-message,──return-code──)──────────►◄
```

WLM_REFRESH option descriptions

WLM-environment
> Specifies the name of the WLM environment that you want to refresh. This is an input parameter of type VARCHAR(32).

ssid
> Specifies the subsystem ID of the DB2 subsystem with which the WLM environment is associated. This is an input parameter of type VARCHAR(4).

status-message
> Contains an informational message about the execution of the WLM refresh. This is an output parameter of type VARCHAR(120).

return-code
> Contains the return code from the WLM_REFRESH call, which is one of the following values:

> **0**   WLM_REFRESH executed successfully.

> **4**   One of the following conditions exists:
>   - The SAF resource profile *ssid*.WLM_REFRESH.*wlm-environment* is not defined in resource class DSNR.
>   - The SQL authorization ID of the process (CURRENT SQLID) is not defined to SAF.

> **8**   The SQL authorization ID of the process (CURRENT SQLID) is not authorized to refresh the WLM environment.

> **990**   DSNTWR received an unexpected SQLCODE while determining the current SQLID.

> **995**   DSNTWR is not running as an authorized program.

> *return-code* is an output parameter of type INTEGER.

Example of WLM_REFRESH invocation

Suppose that you want to refresh WLM environment WLMENV1, which is associated with a DB2 subsystem with ID DSN. Assume that you already have READ access to the DSN.WLM_REFRESH.WLMENV1 SAF profile. The CALL statement for WLM_REFRESH looks like this:

```
strcpy(WLMENV,"WLMENV1");
strcpy(SSID,"DSN");

EXEC SQL CALL SYSPROC.WLM_REFRESH(:WLMENV, :SSID, :MSGTEXT, :RC);
```

For a complete example of setting up access to an SAF profile and calling WLM_REFRESH, see job DSNTEJ6W, which is in data set DSN710.SDSNSAMP.

Appendix J. Summary of changes to DB2 for OS/390 and z/OS Version 7

DB2 for OS/390 and z/OS Version 7 delivers an enhanced relational database server solution for OS/390. This release focuses on greater ease and flexibility in managing your data, better reliability, scalability, and availability, and better integration with the DB2 family.

In Version 7, some utility functions are available as optional products; you must separately order and purchase a license to such utilities. Discussion of utility functions in this publication is not intended to otherwise imply that you have a license to them. See *DB2 Utility Guide and Reference* for more information about utilities products.

Enhancements for managing data

Version 7 delivers the following enhancements for managing data:
- DB2 now collects a comprehensive statistics history that:
  - Lets you track changes to the physical design of DB2 objects
  - Lets DB2 predict future space requirements for table spaces and indexes more accurately and run utilities to improve performance
- Database administrators can now manage DB2 objects more easily and no longer must maintain their utility jobs (even when new objects are added) by using enhancements that let them:
  - Dynamically create object lists from a pattern-matching expression
  - Dynamically allocate the data sets that are required to process those objects
- More flexible DBADM authority lets database administrators create views for other users.
- Enhancements to management of constraints let you specify a constraint at the time you create primary or unique keys. A new restriction on the DROP INDEX statement requires that you drop the primary key, unique key, or referential constraint before you drop the index that enforces a constraint.

Enhancements for reliability, scalability, and availability

Version 7 delivers the following enhancements for the reliability, scalability, and availability of your e-business:
- The DB2 Utilities Suite provides utilities for all of your data management tasks that are associated with the DB2 catalog.
- The new UNLOAD utility lets you unload data from a table space or an image copy data set. In most cases, the UNLOAD utility is faster than the DSNTIAUL sample program, especially when you activate partition parallelism for a large partitioned table space. UNLOAD is also easier to use than REORG UNLOAD EXTERNAL.
- The new COPYTOCOPY utility lets you make additional image copies from a primary image copy and registers those copies in the DB2 catalog. COPYTOCOPY leaves the target object in read/write access mode (UTRW), which allows Structured Query Language (SQL) statements and some utilities to run concurrently with the same target objects.

- Parallel LOAD with multiple inputs lets you easily load large amounts of data into partitioned table spaces for use in data warehouse applications or business intelligence applications. Parallel LOAD with multiple inputs runs in a single step, rather than in different jobs.
- A faster online REORG is achieved through the following enhancements:
  - Online REORG no longer renames data sets, which greatly reduces the time that data is unavailable during the SWITCH phase.
  - Additional parallel processing improves the elapsed time of the BUILD2 phase of REORG SHRLEVEL(CHANGE) or SHRLEVEL(REFERENCE).
- More concurrency with online LOAD RESUME is achieved by letting you give users read and write access to the data during LOAD processing so that you can load data concurrently with user transactions.
- More efficient processing for SQL queries:
  - More transformations of subqueries into a join for some UPDATE and DELETE statements
  - Fewer sort operations for queries that have an ORDER BY clause and WHERE clauses with predicates of the form COL=*constant*
  - More parallelism for IN-list index access, which can improve performance for queries involving IN-list index access
- The ability to change system parameters without stopping DB2 supports online transaction processing and e-business without interruption.
- Improved availability of user objects that are associated with failed or canceled transactions:
  - You can cancel a thread without performing rollback processing.
  - Some restrictions imposed by the restart function have been removed.
  - A NOBACKOUT option has been added to the CANCEL THREAD command.
- Improved availability of the DB2 subsystem when a log-read failure occurs: DB2 now provides a timely warning about failed log-read requests and the ability to retry the log read so that you can take corrective action and avoid a DB2 outage.
- Improved availability in the data sharing environment:
  - Group attachment enhancements let DB2 applications generically attach to a DB2 data sharing group.
  - A new LIGHT option of the START DB2 command lets you restart a DB2 data sharing member with a minimal storage footprint, and then terminate normally after DB2 frees the retained locks that it can.
  - You can let changes in structure size persist when you rebuild or reallocate a structure.
- Additional data sharing enhancements include:
  - Notification of incomplete units of recovery
  - Use of a new OS/390 and z/OS function to improve failure recovery of group buffer pools
- An additional enhancement for e-business provides improved performance with preformatting for INSERT operations.

Easier development and integration of e-business applications

Version 7 provides the following enhancements, which let you more easily develop and integrate applications that access data from various DB2 operating systems and distributed environments:

- DB2 XML Extender for OS/390 and z/OS, a new member of the DB2 Extender family, lets you store, retrieve, and search XML documents in a DB2 database.

- Improved support for UNION and UNION ALL operators in a view definition, a nested table expression, or a subquery predicate, improves DB2 family compatibility and is consistent with SQL99 standards.
- More flexibility with SQL gives you greater compatibility with DB2 on other operating systems:
  - Scrollable cursors let you move forward, backward, or randomly through a result table or a result set. You can use scrollable cursors in any DB2 applications that do not use DB2 private protocol access.
  - A search condition in the WHERE clause can include a subquery in which the base object of both the subquery and the searched UPDATE or DELETE statement are the same.
  - A new SQL clause, FETCH FIRST *n* ROWS, improves performance of applications in a distributed environment.
  - Fast implicit close in which the DB2 server, during a distributed query, automatically closes the cursor when the application attempts to fetch beyond the last row.
  - Support for options USER and USING in a new authorization clause for CONNECT statements lets you easily port applications that are developed on the workstation to DB2 for OS/390 and z/OS. These options also let applications that run under WebSphere to reuse DB2 connections for different users and to enable DB2 for OS/390 and z/OS to check passwords.
  - For positioned updates, you can specify the FOR UPDATE clause of the cursor SELECT statement without a list of columns. As a result, all updatable columns of the table or view that is identified in the first FROM clause of the fullselect are included.
  - A new option of the SELECT statement, ORDER BY *expression*, lets you specify operators as the sort key for the result table of the SELECT statement.
  - New datetime ISO functions return the day of the week with Monday as day 1 and every week with seven days.
- Enhancements to Open Database Connectivity (ODBC) provide partial ODBC 3.0 support, including many new application programming interfaces (APIs), which increase application portability and alignment with industry standards.
- Enhancements to the LOAD utility let you load the output of an SQL SELECT statement directly into a table.
- A new component called Precompiler Services lets compiler writers modify their compilers to invoke Precompiler Services and produce an *SQL statement coprocessor*. An SQL statement coprocessor performs the same functions as the DB2 precompiler, but it performs those functions at compile time. If your compiler has an SQL statement coprocessor, you can eliminate the precompile step in your batch program preparation jobs.
- Support for Unicode-encoded data lets you easily store multilingual data within the same table or on the same DB2 subsystem. The Unicode encoding scheme represents the code points of many different geographies and languages.

Improved connectivity

Version 7 offers improved connectivity:
- Support for COMMIT and ROLLBACK in stored procedures lets you commit or roll back an entire unit of work, including uncommitted changes that are made from the calling application before the stored procedure call is made.

- Support for Windows Kerberos security lets you more easily manage workstation clients who seek access to data and services from heterogeneous environments.
- Global transaction support for distributed applications lets independent DB2 agents participate in a global transaction that is coordinated by an XA-compliant transaction manager on a workstation or a gateway server (Microsoft Transaction Server or Encina, for example).
- Support for a DB2 Connect Version 7 enhancement lets remote workstation clients quickly determine the amount of time that DB2 takes to process a request (the server elapsed time).
- Additional enhancements include:
  - Support for connection pooling and transaction pooling for IBM DB2 Connect
  - Support for DB2 Call Level Interface (DB2 CLI) bookmarks on DB2 UDB for UNIX, Windows, OS/2

Features of DB2 for OS/390 and z/OS

Version 7 of DB2 UDB Server for OS/390 and z/OS offers several features that help you integrate, analyze, summarize, and share data across your enterprise:

- DB2 Warehouse Manager feature. The DB2 Warehouse Manager feature brings together the tools to build, manage, govern, and access DB2 for OS/390-based data warehouses. The DB2 Warehouse Manager feature uses proven technologies with new enhancements that are not available in previous releases, including:
  - DB2 Warehouse Center, which includes:
    - DB2 Universal Database Version 7 Release 1 Enterprise Edition
    - Warehouse agents for UNIX, Windows, and OS/390
    - Information Catalog
  - QMF Version 7
  - QMF High Performance Option
  - QMF for Windows
- DB2 Management Clients Package. The elements of the DB2 Management Clients Package are:
  - DB2 Control Center
  - DB2 Stored Procedure Builder
  - DB2 Installer
  - DB2 Visual Explain
  - DB2 Estimator
- Net Search Extender for in-memory text search for e-business applications
- Net.Data for secure Web applications

Migration considerations

Migration with full fallback protection is available when you have either DB2 for OS/390 and z/OS Version 5 or DB2 for OS/390 and z/OS Version 6 installed. You should ensure that you are fully operational on DB2 for OS/390 and z/OS Version 5, or later, before migrating to DB2 for OS/390 and z/OS Version 7.

To learn about all of the migration considerations from Version 5 to Version 7, read the *DB2 Release Planning Guide* for Version 6 and Version 7; to learn about content information, also read appendixes A through F in both books.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

937

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74/G4
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming interface information

This book is intended to help you to write programs that contain SQL statements. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IBM DATABASE 2 Universal Database Server for OS/390 and z/OS (DB2 for OS/390 and z/OS).

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 for OS/390 and z/OS.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance Information.

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

┌─ **Product-sensitive Programming Interface** ─────────────────

General-use Programming Interface and Associated Guidance Information ...

└─ **End of Product-sensitive Programming Interface** ────────────

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both.

3090	GDDM
	IBM
APL2	IBM Registry
AD/Cycle	IMS
AS/400	IMS/ESA
BookManager	Language Environment
C/370	MVS/DFP
CICS	MVS/ESA
CICS/ESA	Net.Data
CICS/MVS	OpenEdition
COBOL/370	Operating System/390
DATABASE 2	OS/2
DataHub	OS/390
DataPropagator	OS/400
DB2	Parallel Sysplex
DB2 Connect	PR/SM
DB2 Universal Database	QMF
DFSMS/MVS	RACF
DFSMSdfp	RAMAC
DFSMSdss	RETAIN
DFSMShsm	RMF
DFSORT	SAA
Distributed Relational Database Architecture	SecureWay
DRDA	SQL/DS
Enterprise Storage Server	System/370
Enterprise System/3090	System/390
Enterprise System/9000	VisualAge
ES/3090	VTAM
ESCON	

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, other countries, or both.

Java, JDBC, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2. A complete list of DB2 abend reason codes and their explanations is contained in *DB2 Messages and Codes*.

abnormal end of task (abend). Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access path. The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

address space. A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space that contains a task that is connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See also *allied address space* and *task control block*.

after trigger. A trigger that is defined with the trigger activation time AFTER.

agent. As used in DB2, the structure that associates all processes that are involved in a DB2 unit of work. An *allied agent* is generally synonymous with an *allied thread*. *System agents* are units of work that process independently of the allied agent, such as prefetch processing, deferred writes, and service tasks.

alias. An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

allied address space. An area of storage that is external to DB2 and that is connected to DB2. An allied address space is capable of requesting DB2 services.

allied thread. A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

ambiguous cursor. A database cursor that is not defined with the FOR FETCH ONLY clause or the FOR UPDATE OF clause, is not defined on a read-only result table, is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement, and is in a

plan or package that contains either PREPARE or EXECUTE IMMEDIATE SQL statements.

API. Application programming interface.

application. A program or set of programs that performs a task; for example, a payroll application.

application-directed connection. A connection that an application manages using the SQL CONNECT statement.

application plan. The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application process. The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API). A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester. The component on a remote system that generates DRDA requests for data on behalf of an application. An application requester accesses a DB2 database server using the DRDA application-directed protocol.

application server. The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC* and *Unicode*.

attribute. A characteristic of an entity. For example, in database design, the phone number of an employee is one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges is allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

auxiliary index. An index on an auxiliary table in which each index entry refers to a LOB.

auxiliary table. A table that stores columns outside the table in which they are defined. Contrast with *base table*.

B

backout.   The process of undoing uncommitted changes that an application process made. This might be necessary in the event of a failure on the part of an application process, or as a result of a deadlock situation.

base table.   (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

base table space.   A table space that contains base tables.

before trigger.   A trigger that is defined with the trigger activation time BEFORE.

binary integer.   A basic data type that can be further classified as small integer or large integer.

binary large object (BLOB).   A sequence of bytes where the size of the value ranges from 0 bytes to 2 GB–1. Such a string does not have an associated CCSID.

binary string.   A sequence of bytes that is not associated with a CCSID. For example, the BLOB data type is a binary string.

bind.   The process by which the output from the SQL precompiler is converted to a usable control structure, often called an access plan, application plan, or package. During this process, access paths to the data are selected and some authorization checking is performed. The types of bind are:

> **automatic bind**. (More correctly, *automatic rebind*) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.
> **dynamic bind**. A process by which SQL statements are bound as they are entered.
> **incremental bind**. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.
> **static bind**. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

bit data.   Data that is character type CHAR or VARCHAR and is not associated with a coded character set.

BLOB.   Binary large object.

block fetch.   A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch only applies to cursors that do not update data.

BMP.   Batch Message Processing (IMS).

built-in function.   A function that DB2 supplies. Contrast with *user-defined function*.

C

CAF.   Call attachment facility.

call attachment facility (CAF).   A DB2 attachment facility for application programs that run in TSO or MVS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

cast function.   A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

catalog.   In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table.   Any table in the DB2 catalog.

CCSID.   Coded character set identifier.

CDB.   Communications database.

CDRA.   Character data representation architecture.

central processor (CP).   The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

character conversion.   The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA).   An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB).   A sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB–1. In general, character large object values are used whenever a character string might exceed the limits of the VARCHAR type.

character set.   A defined set of characters.

| **character string.** A sequence of bytes that represent
| bit data, single-byte characters, or a mixture of
| single-byte and multibyte characters.

check constraint. See *table check constraint*.

check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain. Check constraints are also called *table check constraints*.

check integrity. The condition that exists when each row in a table conforms to the check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce check constraints on operations that add or change data.

check integrity. The condition that exists when each row in a table conforms to the table check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce table check constraints on operations that add or change data.

check pending. A state of a table space or partition that prevents its use by some utilities and some SQL statements because of rows that violate referential constraints, table check constraints, or both.

CICS. Represents (in this publication) one of the following products:

> **CICS Transaction Server for OS/390**: Customer Information Control System Transaction Server for OS/390
> **CICS/ESA**: Customer Information Control System/Enterprise Systems Architecture
> **CICS/MVS**: Customer Information Control System/Multiple Virtual Storage

CICS attachment facility. A DB2 subcomponent that uses the MVS subsystem interface (SSI) and cross storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

claim. A notification to DB2 that an object is being accessed. Claims prevent drains from occurring until the claim is released, which usually occurs at a commit point. Contrast with *drain*.

claim class. A specific type of object access that can be one of the following:

> Cursor stability (CS)
> Repeatable read (RR)
> Write

claim count. A count of the number of agents that are accessing an object.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

CLOB. Character large object.

clustering index. An index that determines how rows are physically ordered in a table space.

coded character set. A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

coded character set identifier (CCSID). A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs consisting of a character set identifier and an associated code page identifier.

| **code page.** A set of assignments of characters to
| code points. In EBCDIC, for example, the character 'A'
| is assigned code point X'C1', and character 'B' is
| assigned code point X'C2'. Within a code page, each
| code point has only one specific meaning.

| **code point.** In CDRA, a unique bit pattern that
| represents a character in a code page.

collection. A group of packages that have the same qualifier.

column function. An operation that derives its result by using values from one or more rows. Contrast with *scalar function*.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

commit. The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multisite update process that requests all participants to commit the effects of the logical unit of work.

communications database (CDB). A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator. A token (such as =, >, <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

concurrency. The shared use of resources by more than one application process at the same time.

connection. In SNA, the existence of a communication path between two partner LUs that allows information to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

consistency token. A timestamp that is used to generate the version identifier for an application. See also *version*.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

constraint. A rule that limits the values that can be inserted, deleted, or updated in a table. See *referential constraint*, *table check constraint*, and *uniqueness constraint*.

correlated columns. A relationship between the value of one column and the value of another column.

correlated subquery. A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation name. An identifier that designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

CP. See *central processor (CP)*.

created temporary table. A table that holds temporary data and is defined with the SQL statement CREATE GLOBAL TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog, so this kind of table is persistent and can be shared across application processes. Contrast with *declared temporary table*. See also *temporary table*.

CS. Cursor stability.

current data. Data within a host structure that is current with (identical to) the data within the base table.

current SQL ID. An ID that, at a single point in time, holds the privileges that are exercised when certain dynamic SQL statements run. The current SQL ID can be a primary authorization ID or a secondary authorization ID.

cursor sensitivity. The degree to which database updates are visible to the subsequent FETCH statements in a cursor. A cursor can be sensitive to changes that are made with positioned update and delete statements specifying the name of that cursor. A cursor can also be sensitive to changes that are made

with searched update or delete statements, or with cursors other than this cursor. These changes can be made by this application process or by another application process.

cursor stability (CS). The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors.

D

DASD. Direct access storage device.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. A thread that accesses data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA). An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.

database descriptor (DBD). An internal representation of a DB2 database definition, which reflects the data definition that is in the DB2 catalog. The objects that are defined in a database descriptor are table spaces, tables, indexes, index spaces, and relationships.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data stored within it.

database request module (DBRM). A data set member that is created by the DB2 precompiler and that contains information about SQL statements. DBRMs are used in the bind process.

database server. The target of a request from a local application or an intermediate database server. In the DB2 environment, the database server function is provided by the distributed data facility to access DB2 data from local applications, or from a remote database server that acts as an intermediate database server.

DATABASE 2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

data currency. The state in which data that is retrieved into a host variable in your program is a copy of data in the base table.

data definition name (ddname). The name of a data definition (DD) statement that corresponds to a data control block containing the same name.

Data Language/I (DL/I). The IMS data manipulation language; a common high-level interface between a user application and IMS.

data partition. A VSAM data set that is contained within a partitioned table space.

data sharing. The ability of two or more DB2 subsystems to directly access and change a single set of data.

data sharing group. A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member. A DB2 subsystem that is assigned by XCF services to a data sharing group.

data space. A range of up to 2 GB of contiguous virtual storage addresses that a program can directly manipulate. Unlike an address space, a data space can hold only data; it does not contain common areas, system data, or programs.

data type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

date. A three-part value that designates a day, month, and year.

date duration. A decimal integer that represents a number of years, months, and days.

datetime value. A value of the data type DATE, TIME, or TIMESTAMP.

DBA. Database administrator.

DBCLOB. Double-byte character large object.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem allowing a user to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2I. DATABASE 2 Interactive.

DB2I Kanji Feature. The tape that contains the panels and jobs that allow a site to display DB2I panels in Kanji.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

ddname. Data definition name.

deadlock. Unresolvable contention for the use of a resource such as a table or an index.

declarations generator (DCLGEN). A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information. DCLGEN is also a DSN subcommand.

declared temporary table. A table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog, so this kind of table is not persistent and can only be used by the application process that issued the DECLARE statement. Contrast with *created temporary table*. See also *temporary table*.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

delete hole. The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table, because another cursor deleted the row between the time the cursor first included the row in the result table and the time the cursor tried to refetch it.

delete trigger. A trigger that is defined with the triggering SQL operation DELETE.

delimited identifier. A sequence of characters that are enclosed within double quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in syntax diagrams.

dependent. An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See *parent row*, *parent table*, *parent table space*.

deterministic function. A user-defined function whose result is dependent on the values of the input

arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast this with an *not-deterministic function* (sometimes called a *variant function*), which might not always produce the same result for the same inputs.

dimension.   A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. See also *dimension table*.

dimension table.   The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also *dimension*, *star schema*, and *star join*.

direct access storage device (DASD).   A device in which access time is independent of the location of the data.

distinct type.   A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data facility (DDF).   A set of DB2 components through which DB2 communicates with another RDBMS.

Distributed Relational Database Architecture (DRDA).   A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems.

DL/I.   Data Language/I.

double-byte character large object (DBCLOB).   A sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, double-byte character large object values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

double-byte character set (DBCS).   A set of characters, which are used by national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with *single-byte character set* and *multibyte character set*.

drain.   The act of acquiring a locked resource by quiescing access to that object.

drain lock.   A lock on a claim class that prevents a claim from occurring.

DRDA.   Distributed Relational Database Architecture.

DRDA access.   An open method of accessing distributed data that you can use to can connect to another database server to execute packages that were previously bound at the server location. You use the SQL CONNECT statement or an SQL statement with a three-part name to identify the server. Contrast with *private protocol access*.

DSN.   (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration.   A number that represents an interval of time. See *date duration*, *labeled duration*, and *time duration*.

dynamic SQL.   SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

E

EBCDIC.   Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the OS/390, MVS, VM, VSE, and OS/400® environments. Contrast with *ASCII* and *Unicode*.

embedded SQL.   SQL statements that are coded within an application program. See *static SQL*.

equijoin.   A join operation in which the join-condition has the form *expression = expression*.

escape character.   The symbol that is used to enclose an SQL delimited identifier. The escape character is the double quotation mark ("), except in COBOL applications, where the user assigns the symbol, which is either a double quotation mark or an apostrophe (').

EUR.   IBM European Standards.

explicit hierarchical locking.   Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

expression.   An operand or a collection of operators and operands that yields a single value.

external function.   A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function*, *built-in function*, and *SQL function*.

F

false global lock contention. A contention indication from the coupling facility when multiple lock names are hashed to the same indicator and when no real contention exists.

| **fetch orientation.** The specification of the desired
| placement of the cursor as part of a FETCH statement
| (for example, BEFORE, AFTER, NEXT, PRIOR,
| CURRENT, FIRST, LAST, ABSOLUTE, and RELATIVE).

filter factor. A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

foreign key. A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table. Each foreign key value must either match a parent key value in the related parent table or be null.

full outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also *join*.

fullselect. A subselect, a values-clause, or a number of both that are combined by set operators. *Fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

function. A mapping, embodied as a program (the function body), invocable by means of zero or more input values (arguments), to a single value (the result). See also *column function* and *scalar function*.

Functions can be user-defined, built-in, or generated by DB2. (See *built-in function*, *cast function*, *external function*, *sourced function*, *SQL function*, and *user-defined function*.)

function definer. The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function implementer. The authorization ID of the owner of the function program and function package.

function package. A package that results from binding the DBRM for a function program.

function resolution. The process, internal to the DBMS, by which a function invocation is bound to a particular function instance. This process uses the function name, the data types of the arguments, and a

list of the applicable schema names (called the *SQL path*) to make the selection. This process is sometimes called *function selection*.

function selection. See *function resolution*.

G

global lock. A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all the DB2 subsystems of a data sharing group.

global lock contention. Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group name. The MVS XCF identifier for a data sharing group.

group restart. A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

H

help panel. A screen of information presenting tutorial text to assist a user at the terminal.

| **hole.** A row of the result set that cannot be accessed
| because of a delete or update that has been performed
| on the row. See also *delete hole* and *update hole*.

host identifier. A name that is declared in the host program.

host language. A programming language in which you can embed SQL statements.

host program. An application program that is written in a host language and that contains embedded SQL statements.

host structure. In an application program, a structure that is referenced by embedded SQL statements.

host variable. In an application program, an application variable that is referenced by embedded SQL statements.

I

identity column. A column that provides a way for DB2 to automatically generate a numeric value for each row. The generated values are unique if cycling is not used. Identity columns are defined with the AS

IDENTITY clause. Uniqueness of values can be ensured by defining a single-column unique index using the identity column. A table can have no more than one identity column.

IFP.   IMS Fast Path.

IMS.   Information Management System.

IMS attachment facility.   A DB2 subcomponent that uses MVS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

index.   A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

index key.   The set of columns in a table that is used to determine the order of index entries.

index partition.   A VSAM data set that is contained within a partitioning index space.

index space.   A page set that is used to store the entries of one index.

indicator column.   A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable.   A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt.   A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is to be committed or rolled back. At emergency restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be indoubt at restart.

indoubt resolution.   The process of resolving the status of an indoubt logical unit of work to either the committed or the rollback state.

inheritance.   The passing of class resources or attributes from a parent class downstream in the class hierarchy to a child class.

inner join.   The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

inoperative package.   A package that cannot be used because one or more user-defined functions that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with *invalid package.*

insensitive cursor.   A cursor that is not sensitive to inserts, updates, or deletes that are made to the underlying rows of a result table after the result table has materialized.

insert trigger.   A trigger that is defined with the triggering SQL operation INSERT.

Interactive System Productivity Facility (ISPF).   An IBM licensed program that provides interactive dialog services.

inter-DB2 R/W interest.   A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server.   The target of a request from a local application or a remote application requester that is forwarded to another database server. In the DB2 environment, the remote request is forwarded transparently to another database server if the object that is referenced by a three-part name does not reference the local location.

internal resource lock manager (IRLM).   An MVS subsystem that DB2 uses to control communication and database locking.

invalid package.   A package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with *inoperative package.*

IRLM.   Internal resource lock manager.

ISO.   International Standards Organization.

isolation level.   The degree to which a unit of work is isolated from the updating operations of other units of work. See also *cursor stability*, *read stability*, *repeatable read*, and *uncommitted read*.

ISPF.   Interactive System Productivity Facility.

ISPF/PDF.   Interactive System Productivity Facility/Program Development Facility.

J

Japanese Industrial Standards Committee (JISC).   An organization that issues standards for coding character sets.

JCL.   Job control language.

JIS.   Japanese Industrial Standard.

job control language (JCL).   A control language that is used to identify a job to an operating system and to describe the job's requirements.

join.   A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *equijoin, full outer join, inner join, left outer join, outer join, and right outer join.*

K

KB.   Kilobyte (1024 bytes).

key.   A column or an ordered collection of columns identified in the description of a table, index, or referential constraint.

L

labeled duration.   A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

large object (LOB).   A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB–1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

left outer join.   The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also *join*.

linkage editor.   A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit.   The action of creating a loadable computer program using a linkage editor.

L-lock.   Logical lock.

load module.   A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB.   Large object.

LOB locator.   A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB table space.   A table space that contains all the data for a particular LOB column in the related base table.

local.   A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with *remote*.

local lock.   A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem.   The unique RDBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location.   The unique name of a database server. An application uses the location name to access a DB2 database server.

lock.   A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration.   The interval over which a DB2 lock is held.

lock escalation.   The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking.   The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data.

lock mode.   A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object.   The resource that is controlled by a DB2 lock.

lock parent.   For explicit hierarchical locking, a lock that is held on a resource that has child locks that are lower in the hierarchy; usually the table space or partition intent locks are the parent locks.

lock promotion.   The process of changing the size or mode of a DB2 lock to a higher level.

lock size.   The amount of data controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure.   A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

logical index partition.   The set of all keys that reference the same data partition.

logical lock (L-lock).   The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with *physical lock (P-lock)*.

logical unit.   An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW).   The processing that a program performs between synchronization points.

LU name.   Logical unit name, which is the name by which VTAM® refers to a node in a network. Contrast with *location name*.

LUW.   Logical unit of work.

M

mass delete.   The deletion of all rows of a table.

materialize.   (1) The process of putting rows from a view or nested table expression into a work file for additional processing by a query.

(2) The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

MBCS.   Multibyte character set. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2.

menu.   A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

mixed data string.   A character string that can contain both single-byte and double-byte characters.

modify locks.   An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 fails, that DB2 subsystem's modify locks are converted to retained locks.

MPP.   Message processing program (in IMS).

multibyte character set (MBCS).   A character set that represents single characters with more than a single byte. Contrast with *single-byte character set* and *double-byte character set*. See also *Unicode*.

multisite update.   Distributed relational database processing in which data is updated in more than one location within a single unit of work.

MVS.   Multiple Virtual Storage.

MVS/ESA™.   Multiple Virtual Storage/Enterprise Systems Architecture.

N

negotiable lock.   A lock whose mode can be downgraded, by agreement among contending users, to be compatible to all. A physical lock is an example of a negotiable lock.

nested table expression.   A fullselect in a FROM clause (surrounded by parentheses).

nonpartitioning index.   Any index that is not a partitioning index.

nonscrollable cursor.   A cursor that can be moved only in a forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

not-deterministic function.   A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. this type of function is sometimes called a *variant* function. Contrast this with a *deterministic function* (sometimes called a *not-variant function*), which always produces the same result for the same inputs.

not-variant function.   See *deterministic function*.

NUL.   In C, a single character that denotes the end of the string.

null.   A special value that indicates the absence of information.

NUL-terminated host variable.   A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator.   In C, the value that indicates the end of a string. For character strings, the NUL terminator is X'00'.

O

ordinary identifier.   An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token.   A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task.   In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

OS/390.   Operating System/390®.

outer join.   The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also *join*.

overloaded function.   A function name for which multiple function instances exist.

P

package.   An object containing a set of SQL statements that have been statically bound and that is

available for processing. A package is sometimes also called an *application package*.

page. A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB). In a table space, a page contains one or more rows of a table. In a LOB table space, a LOB value can span more than one page, but no more than one LOB value is stored on a page.

page set. Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

panel. A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel task. The execution unit that is dynamically created to process a query in parallel. It is implemented by an MVS service request block.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string were a static SQL statement.

parent row. A row whose primary key value is the foreign key value of a dependent row.

parent table. A table whose primary key is referenced by the foreign key of a dependent table.

parent table space. A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

partitioned page set. A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space. A table space that is subdivided into parts (based on index key range), each of which can be processed independently by utilities.

partner logical unit. An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

path. See *SQL path*.

PCT. Program control table (in CICS).

piece. A data set of a nonpartitioned page set.

physical consistency. The state of a page that is not in a partially changed state.

physical lock (P-lock). A lock type that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with *logical lock (L-lock)*.

physical lock contention. Conflicting states of the requesters for a physical lock. See *negotiable lock*.

plan. See *application plan*.

plan allocation. The process of allocating DB2 resources to a plan in preparation for execution.

plan member. The bound copy of a DBRM that is identified in the member clause.

plan name. The name of an application plan.

P-lock. Physical lock.

point of consistency. A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with *sync point* or *commit point*.

PPT. (1) Processing program table (in CICS). (2) Program properties table (in MVS).

precision. In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 library uses the SQL definitions.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate. An element of a search condition that expresses or implies a comparison operation.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

primary index. An index that enforces the uniqueness of a primary key.

primary key. In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

private connection. A communications connection that is specific to DB2.

private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system. Contrast with *DRDA access*.

private protocol connection. A DB2 private connection of the application process. See also *private connection*.

Q

QMF™.   Query Management Facility.

query block.   The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2's internal processing of the query.

query CP parallelism.   Parallel execution of a single query, which is accomplished by using multiple tasks. See also *Sysplex query parallelism*.

query I/O parallelism.   Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

quiesce point.   A point at which data is consistent as a result of running the DB2 QUIESCE utility.

R

RACF.   Resource Access Control Facility, which is a component of the SecureWay Security Server for OS/390.

RCT.   Resource control table (in CICS attachment facility).

RDB.   Relational database.

RDBMS.   Relational database management system.

RDBNAM.   Relational database name.

read stability (RS).   An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. Under level RS, an application that issues the same query more than once might read additional rows that were inserted and committed by a concurrently executing application process.

rebind.   The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.

record.   The storage representation of a row or other data.

record length.   The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length column.

recovery.   The process of rebuilding databases after a system failure.

referential constraint.   The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential integrity.   The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table upon which the referential constraints are defined.

relational database (RDB).   A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS).   A collection of hardware and software that organizes and provides access to a relational database.

relational database name (RDBNAM).   A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

remote.   Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with *local*.

remote subsystem.   Any RDBMS, except the *local subsystem*, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same MVS system.

reoptimization.   The DB2 process of reconsidering the access path of an SQL statement at run time; during reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

repeatable read (RR).   The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows referenced by the program cannot be changed by other programs until the program reaches a commit point.

request commit.   The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester.   The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource control table (RCT).   A construct of the CICS attachment facility, created by site-provided macro

parameters, that defines authorization and access attributes for transactions or transaction groups.

resource definition online. A CICS feature that you use to define CICS resources online without assembling tables.

resource limit facility (RLF). A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

result set. The set of rows that a stored procedure returns to a client application.

result set locator. A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table. The set of rows that are specified by a SELECT statement.

retained lock. A MODIFY lock that a DB2 subsystem was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 failure.

right outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also *join*.

RLF. Resource limit facility.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

ROWID. Row identifier.

row identifier (ROWID). A value that uniquely identifies a row. This value is stored with the row and never changes.

row lock. A lock on a single row of data.

row trigger. A trigger that is defined with the trigger granularity FOR EACH ROW.

row-value-expression. A comma-separated list of value expressions enclosed in parentheses.

RRSAF. Recoverable Resource Manager Services attachment facility. RRSAF is a DB2 subcomponent that uses OS/390 Transaction Management and Recoverable Resource Manager Services to coordinate resource commitment between DB2 and all other resource managers that also use OS/390 RRS in an OS/390 system.

RS. Read stability.

S

savepoint. A named entity that represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents. The restoration of data and schemas to a savepoint is usually referred to as *rolling back to a savepoint*.

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses. Contrast with *column function*.

scale. In SQL, the number of digits to the right of the decimal point (called the *precision* in the C language). The DB2 library uses the SQL definition.

schema. A logical grouping for user-defined functions, distinct types, triggers, and stored procedures. When an object of one of these types is created, it is assigned to one schema, which is determined by the name of the object. For example, the following statement creates a distinct type T in schema C:

```
CREATE DISTINCT TYPE C.T ...
```

scrollability. The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also *fetch orientation*.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

sensitive cursor. A cursor that is sensitive to changes made to the database after the result table has materialized.

sequential data set. A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

serial cursor. A cursor that can be moved only in a forward direction.

server. The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

share lock. A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with *exclusive lock*.

shift-in character. A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also *shift-out character*.

shift-out character. A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also *shift-in character*.

| **single-byte character set (SBCS).** A set of characters
| in which each character is represented by a single byte.
| Contrast with *double-byte character set* or *multibyte*
| *character set*.

single-precision floating point number. A 32-bit approximate representation of a real number.

size. In the C language, the total number of digits in a decimal number (called the *precision* in SQL). The DB2 library uses the SQL definition.

sourced function. A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *built-in function, external function,* and *SQL function*.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

source type. An existing type that is used to internally represent a distinct type.

space. A sequence of one or more blank characters.

specific function name. A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI. SQL Processor Using File Input.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA. SQL communication area.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

SQLDA. SQL descriptor area.

SQL descriptor area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL/DS. Structured Query Language/Data System. This product is now obsolete and has been replaced by DB2 for VSE & VM.

SQL escape character. The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also *escape character*.

SQL function. A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return only one parameter.

SQL ID. SQL authorization ID.

SQL path. An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the current path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL Processor Using File Input (SPUFI). SQL Processor Using File Input. A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL return code. Either SQLCODE or SQLSTATE.

SQL statement coprocessor. An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

star join. A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also *join*, *dimension*, and *star schema*.

star schema. The combination of a fact table (which contains most of the data) and a number of dimension tables. See also *star join*, *dimension*, and *dimension table*.

statement string. For a dynamic SQL statement, the character string form of the statement.

statement trigger. A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables that are specified by the statement might change).

storage group. A named set of disks on which DB2 data can be stored.

stored procedure. A user-written application program that can be invoked through the use of the SQL CALL statement.

string. See *character string* or *graphic string*.

strong typing. A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

subject table. The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subquery. A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect. That form of a query that does not include ORDER BY clause, UPDATE clause, or UNION operators.

substitution character. A unique character that is substituted during character conversion for any characters in the source program that do not have a match in the target coding representation.

subsystem. A distinct instance of a relational database management system (RDBMS).

sync point. See *commit point*.

synonym. In SQL, an alternative name for a table or view. Synonyms can be used only to refer to objects at the subsystem in which the synonym is defined.

Sysplex query parallelism. Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also *query CP parallelism*.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

system-directed connection. A connection that an RDBMS manages by processing SQL statements with three-part names.

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. See also *base table* or *temporary table*.

table check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

table function. A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator. A mechanism that allows access to trigger transition tables in the FROM clause of SELECT statements, the subselect of INSERT statements, or from within user-defined functions. A table locator is a fullword integer value that represents a transition table.

table space. A page set that is used to store the records in one or more tables.

task control block (TCB). A control block that is used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as one per task), but only one address space connection. See also *address space connection*.

TCB. Task control block (in MVS).

temporary table. A table that holds temporary data; for example, temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two kinds of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with *result table*. See also *created temporary table* and *declared temporary table*.

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name, separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

Time-Sharing Option (TSO). An option in MVS that provides interactive time sharing from remote terminals.

timestamp. A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

TMP. Terminal Monitor Program.

transaction lock. A lock that is used to control concurrent execution of SQL statements.

transition table. A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the new state.

transition variable. A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

trigger. A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table.

trigger activation. The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time. An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true.

trigger cascading. The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action. The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition. An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the *trigger body*.

trigger granularity. A characteristic of a trigger, which determines whether the trigger is activated:
Only once for the triggering SQL statement
Once for each row that the SQL statement modifies

triggering event. The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (INSERT, UPDATE, or DELETE) and a subject table on which the operation is performed.

triggering SQL operation. The SQL operation that causes a trigger to be activated when performed on the subject table.

trigger package. A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

TSO. Time-Sharing Option.

TSO attachment facility. A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker. A parameter marker that is specified along with its target data type. It has the general form:
```
CAST(? AS data-type)
```

type 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*. As of Version 7, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 6 or that are specified as type 2 indexes in Version 4 or later.

V

value. The smallest unit of data that is manipulated in SQL.

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

variant function. See *not-deterministic function*.

varying-length string. A character or graphic string whose length varies within set limits. Contrast with *fixed-length string*.

version. A member of a set of similar programs, DBRMs, packages, or LOBs.
A version of a program is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).

A version of a DBRM is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.

A version of a package is the result of binding a DBRM within a particular database system. The package version is identified by the same program name and consistency token as the DBRM.

A version of a LOB is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.

view.   An alternative representation of data from one or more tables. A view can include all or some of the columns that are contained in tables on which it is defined.

Virtual Storage Access Method (VSAM).   An access method for direct or sequential processing of fixed- and varying-length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

Virtual Telecommunications Access Method (VTAM).   An IBM licensed program that controls communication and the flow of data in an SNA network.

VSAM.   Virtual storage access method.

VTAM.   Virtual Telecommunication Access Method (in MVS).

Z

z/OS.   An operating system for the eServer product line that supports 64-bit real storage.

Bibliography

DB2 Universal Database Server for OS/390 and z/OS Version 7 product libraries:

DB2 for OS/390 and z/OS
- *DB2 Administration Guide, SC26-9931*
- *DB2 Application Programming and SQL Guide, SC26-9933*
- *DB2 Application Programming Guide and Reference for Java, SC26-9932*
- *DB2 Command Reference, SC26-9934*
- *DB2 Data Sharing: Planning and Administration, SC26-9935*
- *DB2 Data Sharing Quick Reference Card, SX26-3846*
- *DB2 Diagnosis Guide and Reference, LY37-3740*
- *DB2 Diagnostic Quick Reference Card, LY37-3741*
- *DB2 Image, Audio, and Video Extenders Administration and Programming, SC26-9947*
- *DB2 Installation Guide, GC26-9936*
- *DB2 Licensed Program Specifications, GC26-9938*
- *DB2 Master Index, SC26-9939*
- *DB2 Messages and Codes, GC26-9940*
- *DB2 ODBC Guide and Reference, SC26-9941*
- *DB2 Reference for Remote DRDA Requesters and Servers, SC26-9942*
- *DB2 Reference Summary, SX26-3847*
- *DB2 Release Planning Guide, SC26-9943*
- *DB2 SQL Reference, SC26-9944*
- *DB2 Text Extender Administration and Programming, SC26-9948*
- *DB2 Utility Guide and Reference, SC26-9945*
- *DB2 What's New? GC26-9946*
- *DB2 XML Extender for OS/390 and z/OS Administration and Programming*, SC27-9949
- *DB2 Program Directory, GI10-8182*

DB2 Administration Tool
- *DB2 Administration Tool for OS/390 and z/OS User's Guide, SC26-9847*

DB2 Buffer Pool Tool
- *DB2 Buffer Pool Tool for OS/390 and z/OS User's Guide and Reference, SC26-9306*

DB2 DataPropagator™
- *DB2 UDB Replication Guide and Reference, SC26-9920*

Net.Data®

The following books are available at this Web site: http://www.ibm.com/software/net.data/library.html
- *Net.Data Library: Administration and Programming Guide for OS/390 and z/OS*
- *Net.Data Library: Language Environment Interface Reference*
- *Net.Data Library: Messages and Codes*
- *Net.Data Library: Reference*

DB2 PM for OS/390
- *DB2 PM for OS/390 Batch User's Guide, SC27-0857*
- *DB2 PM for OS/390 Command Reference, SC27-0855*
- *DB2 PM for OS/390 Data Collector Application Programming Interface Guide, SC27-0861*
- *DB2 PM for OS/390 General Information, GC27-0852*
- *DB2 PM for OS/390 Installation and Customization, SC27-0860*
- *DB2 PM for OS/390 Messages, SC27-0856*
- *DB2 PM for OS/390 Online Monitor User's Guide, SC27-0858*
- *DB2 PM for OS/390 Report Reference Volume 1, SC27-0853*
- *DB2 PM for OS/390 Report Reference Volume 2, SC27-0854*
- *DB2 PM for OS/390 Using the Workstation Online Monitor, SC27-0859*
- *DB2 PM for OS/390 Program Directory, GI10-8223*

Query Management Facility (QMF)
- *Query Management Facility: Developing QMF Applications, SC26-9579*
- *Query Management Facility: Getting Started with QMF on Windows, SC26-9582*
- *Query Management Facility: High Peformance Option User's Guide for OS/390 and z/OS, SC26-9581*
- *Query Management Facility: Installing and Managing QMF on OS/390 and z/OS, GC26-9575*

- *Query Management Facility: Installing and Managing QMF on Windows, GC26-9583*
- *Query Management Facility: Introducing QMF, GC26-9576*
- *Query Management Facility: Messages and Codes, GC26-9580*
- *Query Management Facility: Reference, SC26-9577*
- *Query Management Facility: Using QMF, SC26-9578*

Ada/370
- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer's Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide, SC09-1450*

APL2
- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400

The following books are available at this Web site: www.as400.ibm.com/infocenter
- *DB2 Universal Database for AS/400 Database Programming*
- *DB2 Universal Database for AS/400 Performance and Query Optimization*
- *DB2 Universal Database for AS/400 Distributed Data Management*
- *DB2 Universal Database for AS/400 Distributed Data Programming*
- *DB2 Universal Database for AS/400 SQL Programming Concepts*
- *DB2 Universal Database for AS/400 SQL Programming with Host Languages*
- *DB2 Universal Database for AS/400 SQL Reference*

BASIC
- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

BookManager READ/MVS
- *BookManager READ/MVS V1R3: Installation Planning & Customization, SC38-2035*

SAA® AD/Cycle® C/370
- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1841*

- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User's Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture
- *Character Data Representation Architecture Overview, GC09-2207*
- *Character Data Representation Architecture Reference and Registry, SC09-2190*

CICS/ESA
- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS for MVS/ESA Application Programming Reference, SC33-1170*
- *CICS for MVS/ESA CICS-RACF Security Guide, SC33-1185*
- *CICS for MVS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS for MVS/ESA Customization Guide, SC33-1165*
- *CICS for MVS/ESA Data Areas, LY33-6083*
- *CICS for MVS/ESA Installation Guide, SC33-1163*
- *CICS for MVS/ESA Intercommunication Guide, SC33-1181*
- *CICS for MVS/ESA Messages and Codes, GC33-1177*
- *CICS for MVS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS for MVS/ESA Resource Definition Guide, SC33-1166*
- *CICS for MVS/ESA System Definition Guide, SC33-1164*
- *CICS for MVS/ESA System Programming Reference, GC33-1171*

CICS Transaction Server for OS/390
- *CICS Application Programming Guide, SC33-1687*
- *CICS DB2 Guide, SC33-1939*
- *CICS Resource Definition Guide, SC33-1684*

IBM C/C++ for MVS/ESA
- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*

IBM COBOL

- *IBM COBOL Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-4767*

IBM COBOL for OS/390 & VM Programming Guide, SC26-9049

Conversion Guide
- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment
- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

DataPropagator NonRelational
- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

Data Facility Data Set Services
- *Data Facility Data Set Services: User's Guide and Reference, SC26-4388*

Database Design
- *DB2 Design and Development Guide* by Gabrielle Wiorkowski and David Kull, Addison Wesley, ISBN 0-20158-049-7
- *Handbook of Relational Database Design* by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DataHub®
- *IBM DataHub General Information, GC26-4874*

Data Refresher
- *Data Refresher Relational Extract Manager for MVS* GI10-9927

DB2 Connect®
- *DB2 Connect Enterprise Edition for OS/2 and Windows: Quick Beginnings, GC09-2953*
- *DB2 Connect Enterprise Edition for UNIX: Quick Beginnings, GC09-2952*
- *DB2 Connect Personal Edition Quick Beginnings, GC09-2967*
- *DB2 Connect User's Guide, SC09-2954*

DB2 Red Books
- *DB2 UDB Server for OS/390 Version 6 Technical Update, SG24-6108-00*

DB2 Server for VSE & VM
- *DB2 Server for VM: DBS Utility, SC09-2394*
- *DB2 Server for VSE: DBS Utility, SC09-2395*

DB2 Universal Database for UNIX®, Windows, OS/2
- *DB2 UDB Administration Guide: Planning, SC09-2946*
- *DB2 UDB Administration Guide: Implementation, SC09-2944*
- *DB2 UDB Administration Guide: Performance, SC09-2945*
- *DB2 UDB Administrative API Reference, SC09-2947*
- *DB2 UDB Application Building Guide, SC09-2948*
- *DB2 UDB Application Development Guide, SC09-2949*
- *DB2 UDB CLI Guide and Reference, SC09-2950*
- *DB2 UDB SQL Getting Started, SC09-2973*
- *DB2 UDB SQL Reference Volume 1, SC09-2974*
- *DB2 UDB SQL Reference Volume 2, SC09-2975*

Device Support Facilities
- *Device Support Facilities User's Guide and Reference, GC35-0033*

DFSMS

These books provide information about a variety of components of DFSMS, including DFSMS/MVS®, DFSMSdfp™, DFSMSdss™, DFSMShsm™, and MVS/DFP™.
- *DFSMS/MVS: Access Method Services for the Integrated Catalog, SC26-4906*
- *DFSMS/MVS: Access Method Services for VSAM Catalogs, SC26-4905*
- *DFSMS/MVS: Administration Reference for DFSMSdss, SC26-4929*
- *DFSMS/MVS: DFSMShsm Managing Your Own Data, SH21-1077*
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp, LY27-9606*
- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26–3123*
- *DFSMS/MVS: Macro Instructions for Data Sets, SC26-4913*
- *DFSMS/MVS: Managing Catalogs, SC26-4914*
- *DFSMS/MVS: Program Management, SC26-4916*
- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp, SC26-4920*
- *DFSMS/MVS: Using Advanced Services, SC26-4921*
- *DFSMS/MVS: Utilities, SC26-4926*
- *MVS/DFP: Using Data Sets, SC26-4749*

DFSORT™
- *DFSORT Application Programming: Guide,
  SC33-4035*

Distributed Relational Database Architecture™
- *Data Stream and OPA Reference, SC31-6806*
- *IBM SQL Reference, SC26-8416*
- *Open Group Technical Standard*

  The Open Group presently makes the following
  DRDA books available through its Web site at:
  www.opengroup.org
  - *DRDA Version 2 Vol. 1: Distributed
    Relational Database Architecture (DRDA)*
  - *DRDA Version 2 Vol. 2: Formatted Data
    Object Content Architecture*
  - *DRDA Version 2 Vol. 3: Distributed Data
    Management Architecture*

Domain Name System
- *DNS and BIND, Third Edition, Paul Albitz and
  Cricket Liu, O'Reilly, ISBN 1-56592-512-2*

Education
- *IBM Dictionary of Computing, McGraw-Hill,
  ISBN 0-07031-489-6*
- *1999 IBM All-in-One Education and Training
  Catalog, GR23-8105*

**Enterprise System/9000® and Enterprise
System/3090™**
- *Enterprise System/9000 and Enterprise
  System/3090 Processor Resource/System
  Manager Planning Guide, GA22-7123*

High Level Assembler
- *High Level Assembler for MVS and VM and
  VSE Language Reference, SC26-4940*
- *High Level Assembler for MVS and VM and
  VSE Programmer's Guide, SC26-4941*

Parallel Sysplex® Library
- *OS/390 Parallel Sysplex Application Migration,
  GC28-1863*
- *System/390 MVS Sysplex Hardware and
  Software Migration, GC28-1862*
- *OS/390 Parallel Sysplex Overview: An
  Introduction to Data Sharing and Parallelism,
  GC28-1860*
- *OS/390 Parallel Sysplex Systems Management,
  GC28-1861*
- *OS/390 Parallel Sysplex Test Report,
  GC28-1963*
- *System/390 9672/9674 System Overview,
  GA22-7148*

ICSF/MVS

- *ICSF/MVS General Information, GC23-0093*

IMS
- *IMS Batch Terminal Simulator General
  Information, GH20-5522*
- *IMS Administration Guide: System, SC26-9420*
- *IMS Administration Guide: Transaction
  Manager, SC26-9421*
- *IMS Application Programming: Database
  Manager, SC26-9422*
- *IMS Application Programming: Design Guide,
  SC26-9423*
- *IMS Application Programming: Transaction
  Manager, SC26-9425*
- *IMS Command Reference, SC26-9436*
- *IMS Customization Guide, SC26-9427*
- *IMS Install Volume 1: Installation and
  Verification, GC26-9429*
- *IMS Install Volume 2: System Definition and
  Tailoring, GC26-9430*
- *IMS Messages and Codes, GC27-1120*
- *IMS Utilities Reference: System, SC26-9441*

ISPF
- *ISPF V4 Dialog Developer's Guide and
  Reference, SC34-4486*
- *ISPF V4 Messages and Codes, SC34-4450*
- *ISPF V4 Planning and Customizing, SC34-4443*
- *ISPF V4 User's Guide, SC34-4484*

Language Environment
- *Debug Tool User's Guide and Reference,
  SC09-2137*

National Language Support
- *IBM National Language Support Reference
  Manual Volume 2, SE09-8002*

NetView®
- *NetView Installation and Administration Guide,
  SC31-8043*
- *NetView User's Guide, SC31-8056*

Microsoft® ODBC
- *Microsoft ODBC 3.0 Software Development Kit
  and Programmer's Reference, Microsoft Press,
  ISBN 1-57231-516-4*

OS/390
- *OS/390 C/C++ Programming Guide, SC09-2362*
- *OS/390 C/C++ Run-Time Library Reference,
  SC28-1663*
- *OS/390 C/C++ User's Guide, SC09-2361*
- *OS/390 eNetwork Communications Server: IP
  Configuration, SC31-8513*

- *OS/390 Hardware Configuration Definition Planning, GC28-1750*
- *OS/390 Information Roadmap, GC28-1727*
- *OS/390 Introduction and Release Guide, GC28-1725*
- *OS/390 JES2 Initialization and Tuning Guide, SC28-1791*
- *OS/390 JES3 Initialization and Tuning Guide, SC28-1802*
- *OS/390 Language Environment for OS/390 & VM Concepts Guide, GC28-1945*
- *OS/390 Language Environment for OS/390 & VM Customization, SC28-1941*
- *OS/390 Language Environment for OS/390 & VM Debugging Guide, SC28-1942*
- *OS/390 Language Environment for OS/390 & VM Programming Guide, SC28-1939*
- *OS/390 Language Environment for OS/390 & VM Programming Reference, SC28-1940*
- *OS/390 MVS Diagnosis: Procedures, LY28-1082*
- *OS/390 MVS Diagnosis: Reference, SY28-1084*
- *OS/390 MVS Diagnosis: Tools and Service Aids, LY28-1085*
- *OS/390 MVS Initialization and Tuning Guide, SC28-1751*
- *OS/390 MVS Initialization and Tuning Reference, SC28-1752*
- *OS/390 MVS Installation Exits, SC28-1753*
- *OS/390 MVS JCL Reference, GC28-1757*
- *OS/390 MVS JCL User's Guide, GC28-1758*
- *OS/390 MVS Planning: Global Resource Serialization, GC28-1759*
- *OS/390 MVS Planning: Operations, GC28-1760*
- *OS/390 MVS Planning: Workload Management, GC28-1761*
- *OS/390 MVS Programming: Assembler Services Guide, GC28-1762*
- *OS/390 MVS Programming: Assembler Services Reference, GC28-1910*
- *OS/390 MVS Programming: Authorized Assembler Services Guide, GC28-1763*
- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1764, GC28-1765, GC28-1766, GC28-1767*
- *OS/390 MVS Programming: Callable Services for High-Level Languages, GC28-1768*
- *OS/390 MVS Programming: Extended Addressability Guide, GC28-1769*
- *OS/390 MVS Programming: Sysplex Services Guide, GC28-1771*
- *OS/390 MVS Programming: Sysplex Services Reference, GC28-1772*
- *OS/390 MVS Programming: Workload Management Services, GC28-1773*
- *OS/390 MVS Routing and Descriptor Codes, GC28-1778*
- *OS/390 MVS Setting Up a Sysplex, GC28-1779*
- *OS/390 MVS System Codes, GC28-1780*
- *OS/390 MVS System Commands, GC28-1781*
- *OS/390 MVS System Messages Volume 1, GC28-1784*
- *OS/390 MVS System Messages Volume 2, GC28-1785*
- *OS/390 MVS System Messages Volume 3, GC28-1786*
- *OS/390 MVS System Messages Volume 4, GC28-1787*
- *OS/390 MVS System Messages Volume 5, GC28-1788*
- *OS/390 MVS Using the Subsystem Interface, SC28-1789*
- *OS/390 Security Server External Security Interface (RACROUTE) Macro Reference, GC28-1922*
- *OS/390 Security Server (RACF) Auditor's Guide, SC28-1916*
- *OS/390 Security Server (RACF) Command Language Reference, SC28-1919*
- *OS/390 Security Server (RACF) General User's Guide, SC28-1917*
- *OS/390 Security Server (RACF) Introduction, GC28-1912*
- *OS/390 Security Server (RACF) Macros and Interfaces, SK2T-6700 (OS/390 Collection Kit), SK27-2180 (OS/390 Security Server Information Package)*
- *OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915*
- *OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913*
- *OS/390 SMP/E Reference, SC28-1806*
- *OS/390 SMP/E User's Guide, SC28-1740*
- *OS/390 Support for Unicode: Using Conversion Services, SC33-7050*
- *OS/390 RMF User's Guide, SC28-1949*
- *OS/390 TSO/E CLISTS, SC28-1973*
- *OS/390 TSO/E Command Reference, SC28-1969*
- *OS/390 TSO/E Customization, SC28-1965*
- *OS/390 TSO/E Messages, GC28-1978*
- *OS/390 TSO/E Programming Guide, SC28-1970*
- *OS/390 TSO/E Programming Services, SC28-1971*
- *OS/390 TSO/E REXX Reference, SC28-1975*
- *OS/390 TSO/E User's Guide, SC28-1968*
- *OS/390 DCE Administration Guide, SC28-1584*
- *OS/390 DCE Introduction, GC28-1581*
- *OS/390 DCE Messages and Codes, SC28-1591*

- *OS/390 UNIX System Services Command Reference, SC28-1892*
- *OS/390 UNIX System Services Messages and Codes, SC28-1908*
- *OS/390 UNIX System Services Planning, SC28-1890*
- *OS/390 UNIX System Services User's Guide, SC28-1891*
- *OS/390 UNIX System Services Programming: Assembler Callable Services Reference, SC28-1899*

PL/I for MVS & VM
- *IBM PL/I MVS & VM Language Reference, SC26-3114*
- *IBM PL/I MVS & VM Programming Guide, SC26-3113*

OS PL/I
- *OS PL/I Programming Language Reference, SC26-4308*
- *OS PL/I Programming Guide, SC26-4307*

Prolog
- *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892*

RAMAC® and Enterprise Storage Server™
- *IBM RAMAC Virtual Array, SG24-4951*
- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy, SG24-5338*
- *Enterprise Storage Server Introduction and Planning, GC26-7294*

Remote Recovery Data Facility
- *Remote Recovery Data Facility Program Description and Operations, LY37-3710*

Storage Management
- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *MVS/ESA Storage Management Library: Leading a Storage Administration Group, SC26-3126*
- *MVS/ESA Storage Management Library: Managing Data, SC26-3124*
- *MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125*
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659*

System/370™ and System/390
- *ESA/370 Principles of Operation, SA22-7200*
- *ESA/390 Principles of Operation, SA22-7201*

- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*

System Network Architecture (SNA)
- *SNA Formats, GA27-3136*
- *SNA LU 6.2 Peer Protocols Reference, SC31-6808*
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084*
- *SNA/Management Services Alert Implementation Guide, GC31-6809*

TCP/IP
- *IBM TCP/IP for MVS: Customization & Administration Guide, SC31-7134*
- *IBM TCP/IP for MVS: Diagnosis Guide, LY43-0105*
- *IBM TCP/IP for MVS: Messages and Codes, SC31-7132*
- *IBM TCP/IP for MVS: Planning and Migration Guide, SC31-7189*

VS COBOL II
- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, GC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VS Fortran
- *VS Fortran Version 2: Language and Library Reference, SC26-4221*
- *VS Fortran Version 2: Programming Guide for CMS and MVS, SC26-4222*

VTAM
- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

Special Characters

: (colon)
  assembler host variable   111
  C host variable   124
  C program   136
  COBOL   146
  COBOL host variable   146
  FORTRAN   167
  FORTRAN host variable   167
  PL/I host variable   178
  preceding a host variable   68
_ (underscore)
  assembler host variable   109

A

abend
  DB2   736, 769
  effect on cursor position   91
  exit routines   751
  for synchronization calls   480
  IMS
    U0102   485
    U0775   363
    U0778   364
  multiple-mode program   359
  program   358
  reason codes   752
  return code posted to CAF CONNECT   739
  return code posted to RRSAF CONNECT   771
  single-mode program   359
  system
    X'04E'   477
ABRT parameter of CAF (call attachment facility)   745, 754
access path
  affects lock attributes   351
  direct row access   677
  index-only access   677
  low cluster ratio
    suggests table space scan   684
    with list prefetch   703
  multiple index access
    description   687
    PLAN_TABLE   676
  selection
    influencing with SQL   655
    problems   621
    queries containing host variables   644
    Visual Explain   655, 667
  table space scan   684
  unique index with matching value   689
ACQUIRE
  option of BIND PLAN subcommand
    locking tables and table spaces   337
activity sample table   811
Ada language   66

address space
  initialization
    CAF CONNECT command   741
    CAF OPEN command   743
  sample scenarios   750, 793
  separate tasks   730, 763
  termination
    CAF CLOSE command   745
    CAF DISCONNECT command   747
ALL
  quantified predicate   45
ALLOCATE CURSOR statement
  usage   600
ambiguous cursor   347
AMODE link-edit option   409, 459
ANY
  quantified predicate   45
APL2 application program   66
APOST option
  precompiler   400
apostrophe
  option   400
  string delimiter precompiler option   400
APOSTSQL option
  precompiler   400
application plan
  binding   412
  creating   409
  dynamic plan selection for CICS applications   420
  invalidated
    conditions for   320
  listing packages   412
  rebinding
    changing packages   319
  using packages   315
application program
  coding SQL statements
    assembler   107
    COBOL   141
    data declarations   95
    data entry   26
    description   65
    dynamic SQL   495, 524
    FORTRAN   164
    host variables   68
    PL/I   174
    selecting rows using a cursor   81
  design considerations
    bind   313
    checkpoint   479
    IMS calls   479
    planning for changes   317
    precompile   313
    programming for DL/I batch   478
    SQL statements   479
    stored procedures   525
    structure   725
    symbolic checkpoint   479

E

S

Readers' Comments — We'd Like to Hear from You

DB2 Universal Database for OS/390 and z/OS
Application Programming
and SQL Guide
Version 7

Publication No. SC26-9933-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?   ☐ Yes   ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

IBM ®

Program Number: 5675-DB2