IBM DB2 Universal Database

Embedded SQL Programming Guide

Version 5

S10J-8158-00

IBM DB2 Universal Database

Embedded SQL Programming Guide

Version 5

S10J-8158-00

Before using this information and the product it supports, be sure to read the general information under Appendix G, “Notices” on page 583.

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties and any statements provided in this manual should not be interpreted as such.

Order publications through your IBM representative or the IBM branch office serving your locality or by calling `1-800-879-2755` in U.S. or `1-800-IBM-4YOU` in Canada.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About This Book

This book discusses how to design and code application programs that access DB2 databases and application servers. It presents detailed information on the use of Structured Query Language (SQL) in supported host language programs. Where you require information unique to your specific operating system, see the appropriate book from the following list. This book refers to the following titles collectively as *DB2 SDK Building Applications*:

- *Building Applications for UNIX Environments*
- *Building Applications for Windows and OS/2 Environments*
- *DB2 SDK for Macintosh Building Your Applications*

To effectively utilize the information in this book to design, write, and test your DB2 application programs, you need to refer to the *SQL Reference* alongside this book. If you are using the DB2 Call Level Interface (CLI) in your applications to access DB2 databases, refer to the *CLI Guide and Reference*. If you want to use the DB2 APIs in your application programs, refer to the *API Reference*. To help you decide which DB2 programming interface or programming method meets your requirements, refer to the *Road Map to DB2 Programming*.

You can access data using embedded SQL statements, (as discussed in this book), or the DB2 Call Level Interface (DB2 CLI) as discussed in the *CLI Guide and Reference*. DB2 CLI may provide some data access capabilities that may not be available through embedded SQL. Capabilities such as the use of scrollable cursors and the use of multiple result sets in stored procedure applications are only available with DB2 CLI. See the discussions in the *Road Map to DB2 Programming* and in “The DB2 Call Level Interface (CLI)” on page 127 to help you decide whether to use embedded SQL or DB2 CLI.

Your programs can perform database manager administration tasks by calling the DB2 Application Programming Interfaces (APIs). These APIs are discussed in detail in the *API Reference*.

Applications can also be developed where one part of the application runs at the client and another part runs at the server. This technique is discussed in Chapter 5, “Writing Stored Procedures” on page 167.

There are object-based extensions to DB2 that you can use to enhance your DB2 application programs by making them more powerful, flexible, and active than traditional DB2 applications. The extensions include: large objects, user-defined distinct types, user-defined functions, and triggers. These features of DB2 are described in Chapter 6, “Using the Object-Relational Capabilities” on page 229, Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285, and Chapter 8, “Using the Active DBMS Capabilities” on page 349.

References to 'DB2' in this book should be understood to mean the 'DB2 Universal Database' product. References to DB2 on other platforms use the specific product name (such as DB2 for MVS/ESA or DB2 for AS/400).

Who Should Use This Book

This book is intended for programmers who are experienced with SQL, and one or more of the supported programming languages.

How To Use This Book

This book is organized, by task, into the following chapters:

- Chapter 1, Getting Started With DB2 Application Development, introduces the overall application development process using DB2 Universal Database. It presents basic information such as the basic programming model and other fundamental concepts to help you begin programming on DB2. In addition, it discusses the main choices you have for coding your applications, and the important application design issues you need to consider prior to coding your applications. This material concludes with information to help you set up a test environment where you can begin to develop your applications.

  Read this information if you are new to programming with DB2, or if you want to review the generalities of DB2 programming.

- Chapter 2, Writing Embedded Static Programs, discusses the details of coding your DB2 application using embedded static SQL. It contains detailed guidelines and considerations for using static SQL.

  Read this information if you have decided after reading Chapter 1 that you want to use static SQL in your applications.

- Chapter 3, Writing Dynamic Programs, discusses the details of coding your DB2 application using dynamic SQL. It contains detailed guidelines and considerations for using dynamic SQL.

  Read this information if you have decided after reading Chapter 1 that you want to use dynamic SQL in your applications.

- Chapter 4, Programming Considerations for Concurrency and Performance, discusses topics and programming techniques that you should consider in order to develop efficient DB2 application programs. Topics discussed include: concurrency, locking, row blocking, performance guidelines, the use of the system catalog tables, and more.

  Read this information if you are interested in enhancing the efficiency of your DB2 applications.

- Chapter 5, Writing Stored Procedures, describes how you can use stored procedures for your database manager applications running in client/server environments.

  Read this information if you are interested in distributed applications for DB2 using stored procedures.

- Chapter 6, Using the Object-Relational Capabilities, describes how you can use the object-oriented capabilities of DB2. Included are the details of extending your traditional application to one that takes advantage of DB2 capabilities such as large objects, user-defined functions, and user-defined distinct types in an object-oriented context.

  Read this information if you want to enhance your database application using the object-oriented features of DB2.

- Chapter 7, Writing User-Defined Functions (UDFs), describes how you write user-defined functions to extend your DB2 applications. Included are the details of the process of writing a user-defined function, programming considerations related to user-defined functions, and several examples that show you how to exploit this important capability. In addition, this chapter describes user-defined table functions, and OLE (Object Linking and Embedding) automation UDFs.

  Read this information if you want to enhance your database application using the user-defined function capability of DB2.

- Chapter 8, Using the Active DBMS Capabilities, describes how you can use triggers to make your database applications active relative to traditional database applications. It also shows how you can enforce business rules within your database applications using the above capabilities.

  Read this information if you want to enhance your database application using triggers.

- Chapter 9, Programming in Complex Environments, describes advanced programming topics such as national language support, dealing with Extended UNIX** Code (EUC) code pages for databases and applications, accessing multiple databases within a unit of work, and multi-threaded applications.

  Read this information if you need to consider any of the above advanced topics in your database application.

- Chapter 10, Programming Considerations in a Partitioned Environment, describes programming considerations if you are developing applications that run in a partitioned environment.

  Read this information if you are developing database applications that run in such an environment.

- Chapter 11, Programming in C and C++, discusses host language specific information concerning database manager applications written in C and C++.
- Chapter 12, Programming in COBOL, discusses host language specific information concerning database manager applications written in COBOL.
- Chapter 13, Programming in FORTRAN, discusses host language specific information concerning database manager applications written in FORTRAN.
- Chapter 14, Programming in REXX, discusses host language specific information concerning database manager applications written in REXX.
- Chapter 15, Programming in Java, discusses host language specific information concerning database manager applications written in Java.

- Appendix A, Supported SQL Statements (DB2 Universal Database), lists the SQL statements supported by DB2 Universal Database.
- Appendix B, Sample Programs and Extra Examples, contains information on supplied example programs in various supported host languages and describes how they work.
- Appendix C, Programming in a DRDA Environment, describes programming considerations for DB2 Connect if you access DRDA databases in your applications in a distributed environment.
- Appendix D, Country Code and Code Page Support, lists the languages and code sets supported by DB2 Universal Database servers.
- Appendix E, Simulating EBCDIC Binary Collation, describes how to collate DB2 character strings according to an EBCDIC, or user-defined, collating sequence.
- Appendix F, How the DB2 Library Is Structured, shows you where you can get more information for the DB2 Universal Database product.

Highlighting Conventions

This book uses the following conventions:

Italics Indicates one of the following:

- Introduction of a new term
- Names or values that are supplied by the user
- Reference to another source of information
- General emphasis.

UPPERCASE Indicates one of the following:

- API names
- Commands
- Database manager data types
- Field names
- Keywords.
- SQL statements.

`Example` Indicates one of the following:

- Coding examples and code fragments
- Examples of output, similar to what is displayed by the system
- Examples of specific data values
- Examples of system messages
- File and directory names
- Information that you are instructed to type.

Bold Bold text is used to emphasize a point.

Related Publications

The following manuals describe how to develop applications for international use and for specific countries:

Form Number	Book Title
SE09-8001	*National Language Design Guide, Volume 1, Designing Enabled Products, Rules and Guidelines*
SE09-8002	*National Language Design Guide, Volume 2, NLS Reference Manual*

The following manual describes how to access different database systems.

Form Number	Book Title
SC26-8416	*IBM SQL Reference (Volumes 1 and 2)*[1]

[1] This two volume set replaces the Formal Register of Extensions and Differences in SQL, SC26-3316. The IBM SQL Reference provides a list of SQL extensions and incompatibilities among various standards and the IBM family of relational database products (including DB2 Universal Database).

Chapter 1. Getting Started With DB2 Application Development

This chapter discusses the process by which a database manager application is developed, from creating the working environment with a sample database, through designing, coding, and running the program.

The specific topics discussed are:

- Prerequisites for Programming
- Coding a DB2 Application
- Designing an Application For DB2
- Alternatives for Coding DB2 Applications
- Creating and Preparing the Source Files
- Creating Packages for Compiled Applications
- Supported SQL Statements
- Authorization Considerations
- Database Manager APIs Used in Embedded or CLI Programs
- Setting Up the Testing Environment
- Running, Testing and Debugging Your Programs
- Prototyping Your SQL Statements

Prerequisites for Programming

The application development process assumes that the appropriate operating environment has been established. This means that the following conditions must exist:

- In a client/server environment only:
  - The database manager product has been installed on the server.
  - The required DB2 SDK software has been installed on the client or server workstation where you are developing your applications.
- A supported compiler has been installed and configured.
- In a client/server environment, a communication protocol that is common to client and server has been installed and configured.
- You are able to connect to the required database using the command line processor.
- Ensure that your profile is set up so you can connect to the correct database instance for the application that you are developing.

For details on how to accomplish these tasks, see the *DB2 SDK Building Applications* and *Quick Beginnings* books for your operating environment.

You can *develop* applications at the server, or on any client that has the DATABASE 2 Software Developer's Kit (DB2 SDK) installed. You can *run* applications either at the server, or on any client that has the Client Application Enabler (CAE) installed. That is, clients that will be executing application programs, but not developing them, do not require the DB2 SDK installed, but must have the CAE installed.

DB2 supports the C, C++, COBOL, and FORTRAN programming languages through its precompilers. In addition, DB2 supports the REXX language through a dynamic interpreter, as well as the Java language. For information on the specific precompilers provided by DB2, and the languages supported on your platform, refer to the *DB2 SDK Building Applications* book for your operating system.

There may be programming capabilities that are provided by other products from IBM, or from non-IBM vendors, that enable you to do such things as use PL/I and BASIC compilers. These capabilities are not specifically supported or provided by DB2. For more information on using these capabilities, see the documentation for the specific product of interest.

DB2 provides a sample database which you require when running the supplied sample applications. For information about the sample database and its contents, see the *SQL Reference*.

Coding a DB2 Application

This section is intended for programmers who are new to developing a DB2 application. It presents a model of the logical parts of a DB2 application. If you are an experienced SQL developer, you can ignore this section.

A DB2 application program is made up of several main parts: setup, connecting to the database, one or more *transaction*s, disconnecting from the database, and ending the program. A *transaction* is a set of database operations that must conclude successfully before being committed to the database. With embedded SQL, a transaction begins implicitly and ends when the application executes either a COMMIT or ROLLBACK statement. An example of a transaction is the entry of a customer’s deposit, and the updating of the customer’s balance.

Certain SQL statements must appear at the beginning and end of the program to handle the transition from the host language to the embedded SQL statements.

The beginning of every program must contain:

- Declarations of all variables and data structures that the database manager uses to interact with the host program.
- SQL statements that provide for error handling by setting up the SQL communications area.

The body of every program contains the SQL statements which access and manage data. These statements constitute transactions. Among the statements included in this section are:

- The CONNECT statement, which establishes a connection to a database server
- Data manipulation statements (for example, the SELECT statement)
- Data definition statements (for example, the CREATE statement)
- Data control statements (for example, the GRANT statement).
- Either the COMMIT or ROLLBACK statement to end a transaction.

The end of the application program typically contains SQL statements that:

- Release the program’s connection to the database server
- Cleanup any resource.

Setting Up the Program

To set up a DB2 program, you typically declare the program variables used and declare the SQL Communications area.

Declaring Variables That Interact with the Database Manager

All host program variables that interact with the database manager must be declared in an SQL declare section. An SQL declare section is a group of host program variable declarations that are preceded by the SQL statement `BEGIN DECLARE SECTION` and followed by the SQL statement `END DECLARE SECTION`. Host program variables declared in an SQL declare section are called *host variables*, and can be used in *host-variable* references in SQL statements. (Here, *host-variable* is a tag used in syntax diagrams in the *SQL Reference*.) A program may contain multiple SQL declare sections.

The attributes of each host variable depend on how the variable is used in the SQL statement. For example, variables that receive data from or store data in DB2 tables must have data type and length attributes compatible with the column being accessed. To determine the data type for each variable, you must be familiar with DB2 data types which are explained in “Data Types” on page 52. Each column of every table is assigned a data type when the table is created.

Relating Host Variables to an SQL Statement

Host variables can be used to receive data from the database manager or to transfer data to it from the host program. Host variables that receive data from the database manager are *output host variables*, while those that transfer data to it from the host program are *input host variables*.

Consider the following SELECT INTO statement:

```
SELECT HIREDATE, EDLEVEL
  INTO :hdate, :lvl
  FROM EMPLOYEE
  WHERE EMPNO = :idno
```

It contains two output host variables, `hdate` and `lvl`, and one input host variable, `idno`. The database manager uses the data stored in the host variable `idno` to determine the EMPNO of the row that is retrieved from the EMPLOYEE table. If a row that meets the search criteria is found, `hdate` and `lvl` receive the data stored in the columns HIREDATE and EDLEVEL, respectively. This statement illustrates an interaction between the host program and the database manager using columns of the EMPLOYEE table.

Each column of a table is assigned a data type, and each data type can be related to a host language data type. For example, the INTEGER data type is a 32-bit signed integer. This is equivalent to the following data description entries in each of the host languages, respectively:

C/C++:

```
long variable_name;
```

COBOL:

```
01  variable-name  PICTURE S9(9) COMPUTATIONAL-5.
```

FORTRAN:

```
INTEGER*4 variable_name
```

For the list of supported SQL data types and the corresponding host language data types, see the following:

- for C/C++, “Supported SQL Data Types” on page 445
- for COBOL, “Supported SQL Data Types” on page 468
- for FORTRAN, “Supported SQL Data Types” on page 485
- for REXX, “Supported SQL Data Types” on page 497.

In order to determine exactly how to define the host variable for use with a column, you need to find out what SQL data type that column has. Do this by querying the system catalog, which is a set of views containing information about all tables created in the database. This catalog is described in the *SQL Reference*.

After you have determined the data types, you can refer to the conversion charts in the host language chapters, and code the appropriate declarations. Figure 6 on page 49 shows examples of declarations in the supported host languages. Note that REXX applications do not need to declare host variables except for LOB locators and file reference variables. Other host variable data types and sizes are determined at run time based on the contents of the variable.

Figure 6 also shows the BEGIN and END DECLARE SECTION statements. Observe how the delimiters for SQL statements differ for each language. For the exact rules of placement, continuation, and delimiting of these statements, see the language-specific chapters of this book.

Handling Errors and Warnings with the SQLCA

The SQL Communications Area (SQLCA) is discussed in detail later in this chapter. This section presents an overview. To declare the SQLCA, code the `INCLUDE SQLCA` statement in your program. For C or C++ applications use:

```
EXEC SQL INCLUDE SQLCA;
```

For COBOL applications use:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

For FORTRAN applications use:

```
EXEC SQL INCLUDE SQLCA
```

When you preprocess your program, the database manager inserts host language variable declarations in place of the INCLUDE SQLCA statement. The system communicates with your program using the variables for warning flags, error codes, and diagnostic information.

After executing each SQL statement, the system returns a return code in both SQLCODE and SQLSTATE. SQLCODE is an integer value that summarizes the execution of the statement, and SQLSTATE is a character field that provides common error codes across IBM's relational database products. SQLSTATE also conforms to the ISO/ANSI SQL92 or FIPS 127-2[2] standard.

Note that if SQLCODE is less than 0, it means an error has occurred and the statement has not been processed. If the SQLCODE is greater than 0, it means a warning has been issued, but the statement is still processed. See the *Message Reference* for a listing of SQLCODE and SQLSTATE error conditions.

If you want the system to control error checking after each SQL statement, use the WHENEVER statement. The following WHENEVER statement indicates to the system what to do when it encounters a negative SQLCODE:

```
WHENEVER SQLERROR GO TO errchk
```

That is, whenever an SQL error occurs, program control is transferred to code that follows the label, such as `errchk`. This code should include logic to analyze the error indicators in the SQLCA. Depending upon the ERRCHK definition, action may be taken to execute the next sequential program instruction, to perform some special functions, or as in most situations, to roll back the current *transaction* and terminate the program. See "Coding Transactions" on page 7 for more information on a transaction and "Diagnostic Handling and the SQLCA Structure" on page 83 for more information about how to control error checking in your application program.

Exercise caution when using the WHENEVER SQLERROR statement. If your application's error handling code contains SQL statements, and if these statements result in an error while processing the original error, your application may enter an infinite loop. This situation is difficult to troubleshoot. The first statement in the destination of a WHENEVER SQLERROR should be WHENEVER SQLERROR CONTINUE. This statement resets the error handler. After this statement, you can safely use SQL statements.

For a DB2 application written in C or C++, if the application is made up of multiple source files, only one of the files should include the EXEC SQL INCLUDE SQLCA

[2] FIPS 127-2 refers to *Federal Information Processing Standards Publication 127-2 for Database Language SQL*. ISO/ANSI SQL92 refers to *American National Standard Database Language SQL X3.135-1992* and *International Standard ISO/IEC 9075:1992, Database Language SQL*.

statement to avoid multiple definitions of the SQLCA. The remaining source files should use the following lines:

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

If your application must be compliant with the ISO/ANSI SQL92 or FIPS 127-2[2] standard, do not use the above statements or the `INCLUDE SQLCA` statement. For the alternative to coding the above statements, refer to “SQLSTATE and SQLCODE Variables” on page 450 for C or C++ applications, “SQLSTATE and SQLCODE Variables” on page 472 for COBOL applications, or “SQLSTATE and SQLCODE Variables” on page 487 for FORTRAN applications.

Using Additional Nonexecutable Statements

Generally, other *nonexecutable* SQL statements are also part of this section of the program. These are discussed later in this manual, and in the *SQL Reference*. Examples of nonexecutable statements are:

- INCLUDE `text-file-name`
- INCLUDE SQLDA
- DECLARE CURSOR

Connecting to the Database Server

Your program must establish a connection to the target database server before it can run any executable SQL statements. This connection identifies the authorization ID of the user who is running the program, and the name of the database server on which the program is run. Generally, your application process can only connect to one database server at a time. This server is called the *current server*. However, your application can connect to multiple database servers within a distributed unit of work (DUOW). In this case, only one server can be the current server. For information on distributed unit of work, see “Distributed Unit of Work” on page 389.

Your program can establish a connection to a database server either explicitly using the CONNECT statement, or implicitly to the default database server. See the *SQL Reference* for a discussion about how to use this statement, and about connection states. A default database server is established when the application requester is initialized. If implicit connect is available and an application process is started, it is implicitly connected to the default database server. It is good practice for the first SQL statement executed by an application program to be the CONNECT statement, to avoid accidentally executing SQL statements against the default database.

After the connection has been established, your program can issue SQL statements that manipulate data, define and maintain database objects, and initiate control operations, such as granting user authority, or committing changes to the database. A connection lasts until a CONNECT RESET, CONNECT TO, or DISCONNECT statement is issued. In a DUOW environment, a connection also lasts until a DB2 RELEASE then DB2 COMMIT is issued. A CONNECT TO statement does not terminate a connection when using DUOW (see “Distributed Unit of Work” on page 389).

Coding Transactions

A transaction is a sequence of SQL statements (possibly with intervening host language code) that the database manager treats as a whole. An alternative term that is often used for transaction is *unit of work*.

The system ensures the consistency of data at the transaction level, by ensuring that either *all* operations within a transaction are completed, or *none* are completed. Suppose, for example, that money is to be deducted from one account and added to another. If both these updates are placed in a single transaction, and if a system failure occurs while they are in progress, then when the system is restarted, the data is automatically restored to the state it was in before the transaction began. If a program error occurs, all changes made by the statement in error are restored. Work done in the transaction prior to execution of the statement in error is not undone, unless you specifically roll it back.

You can code one or more transactions within a single application program; and it is possible to access more than one database from within a single transaction. A transaction that accesses more than one database is called a distributed unit of work (DUOW). For information on these topics, see “Remote Unit of Work” on page 389 and “Distributed Unit of Work” on page 389.

Beginning a Transaction

A transaction begins implicitly with the first *executable* SQL statement and ends by either a COMMIT or a ROLLBACK statement, or when the program ends.

The following are examples of statements that do **not** start a transaction because they are not executable statements:

```
BEGIN DECLARE SECTION          INCLUDE SQLCA
END DECLARE SECTION            INCLUDE SQLDA
DECLARE CURSOR                 WHENEVER
```

An executable SQL statement always occurs within a transaction. If such a statement is encountered after you end a transaction, it automatically starts another.

Ending a Transaction

To end a transaction, you can use either the COMMIT statement to save its changes, or the ROLLBACK statement to ensure that these changes are not saved.

Using the COMMIT Statement: This statement ends the current transaction. It makes the database changes made during the current transaction visible to other processes.

Changes should be committed as soon as application requirements permit. In particular, write your programs so that uncommitted changes are not held while waiting for input from a terminal as this can result in database resources being held for a long time. Holding these resources prevents other applications that need these resources from running.

Your application programs should explicitly end any transactions prior to terminating. If you do not end transactions explicitly, DB2 Universal Database automatically commits all the changes made during the program's pending transaction when the program ends successfully, except on the Windows 95 and Windows NT operating systems. DB2 Universal Database rolls back the changes under the following conditions:

- A log full condition.
- Some other system condition that causes database manager processing to end.

On the Windows 95 and Windows NT operating systems, if you do not explicitly commit the transaction, the database manager always rolls back the changes.

See "Ending the Program" and "Diagnostic Handling and the SQLCA Structure" on page 83 for more information about program termination.

Note: The COMMIT statement has no effect on the contents of host variables.

Using the ROLLBACK Statement: This statement ends the current transaction, and restores the data to the state it was in prior to the transaction beginning.

Notes:

1. The ROLLBACK statement has no effect on the contents of host variables.
2. If you use a ROLLBACK statement in a routine that was entered because of an error or warning and you use the SQL WHENEVER statement, then you should specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK. This avoids a program loop if the ROLLBACK fails with an error or warning.

The ROLLBACK statement should not be issued if a severe error occurs such as the loss of communications between the client and server applications, or if the database gets corrupted. You will receive a message indicating that you cannot issue a ROLLBACK statement in the event of a severe error. The only statement that can be issued after a severe error is a CONNECT statement.

Ending the Program

To properly end your program:

1. End the current transaction (if one is in progress) by explicitly issuing either a COMMIT statement or a ROLLBACK statement.
2. Release your connection to the database server by using the CONNECT RESET statement.
3. Clean up resources used by the program. For example, free any temporary storage or data structures that are used.

Note: If the current transaction is still active when the program terminates, DB2 implicitly ends the transaction. Since DB2's behavior when it implicitly ends a transaction is platform specific, you should explicitly end all transactions by issuing a COMMIT or a ROLLBACK statement before the program terminates.

See Implicitly Ending a Transaction for details of how DB2 implicitly ends a transaction.

Implicitly Ending a Transaction

If your program terminates without ending the current transaction, DB2 implicitly ends the current transaction (see “Ending the Program” on page 8 for details on how to properly end your program). DB2 implicitly terminates the current transaction by issuing either a COMMIT or a ROLLBACK statement when the application ends. Whether a COMMIT or ROLLBACK is issuing by DB2 depends on a number of factors such as:

- Whether the application terminated normally
- The platform on which the DB2 server runs
- Whether the application uses the context APIs (see “Multiple Thread Database Access” on page 395).

On Most Supported Operating Systems

DB2 implicitly commits a transaction if the termination is normal, or implicitly rolls back the transaction if it is abnormal. Note that what your program considers to be an abnormal termination may not be considered abnormal by the database manager. For example, you may code `exit(-16)` when your application encounters an unexpected error and terminate your application abruptly. The database manager considers this to be a normal termination and commits the transaction. The database manager considers items such as an exception or a segmentation violation as abnormal terminations.

On Windows 95 and Windows NT Operating Systems

DB2 always rolls back the transaction regardless of whether your application terminates normally or abnormally, unless you explicitly commit the transaction using the COMMIT statement.

When Using the DB2 Context APIs

Your application can use any of the DB2 APIs to set up and pass application contexts between threads as described in “Multiple Thread Database Access” on page 395. If your application uses these DB2 APIs, DB2 implicitly rolls back the transaction regardless of whether your application terminates normally or abnormally. The transaction is rolled back unless you explicitly commit the transaction using the COMMIT statement.

Summary

Figure 1 summarizes the general framework for a DB2 application program in pseudocode format. This framework must, of course, be tailored to suit your own program.

```
Start Program                                               ┐
EXEC SQL BEGIN DECLARE SECTION                              │
  DECLARE USERID FIXED CHARACTER (8)                        │
  DECLARE PW FIXED CHARACTER (8)                            │
      ●                                                     │  Application
      ●                                                     │  Setup
  (other host variable declarations)                        │
      ●                                                     │
      ●                                                     │
EXEC SQL END DECLARE SECTION                                │
EXEC SQL INCLUDE SQLCA                                      │
EXEC SQL WHENEVER SQLERROR GOTO ERRCHK                      │
      ●                                                     ┘
      ●
      ●                                                     ┐
EXEC SQL CONNECT TO database A USER :userid USING :pw       │
EXEC SQL SELECT ...                                         │
EXEC SQL INSERT ...                                         │  First Unit
      ●                                                     │  of Work
EXEC SQL COMMIT                                             │
      ●                                                     ┘
      ●
      ●                                                     ┐
EXEC SQL CONNECT TO database B USER :userid USING :pw       │
EXEC SQL SELECT ...                                         │
EXEC SQL DELETE ...                                         │  Second Unit
      ●                                                     │  of Work
EXEC SQL COMMIT                                             │
      ●                                                     ┘
      ●
      ●                                                     ┐
EXEC SQL CONNECT TO database A                              │
EXEC SQL SELECT ...                                         │
EXEC SQL DELETE ...                                         │  Third Unit
      ●                                                     │  of Work
EXEC SQL COMMIT                                             │
      ●                                                     ┘
      ●
      ●                                                     ┐
EXEC SQL CONNECT RESET                                      │
ERRCHK                                                      │
      ●                                                     │  Application
  (check error information in SQLCA)                        │  Cleanup
      ●                                                     │
End Program                                                 ┘
```

Figure 1. Pseudocode Framework for Coding Programs

Designing an Application For DB2

DB2 provides you with a variety of application development capabilities that you can use to supplement or extend the traditional capabilities of an application. As an application designer, the design decision that you must make is the most fundamental one: *which DB2 capabilities should I use in the design of my application?* In order to make appropriate choices, you need to consider both the database design and target environments for your application. For example, you can choose to include some of the enforcement of business rules in the database design instead of including the logic in your application.

The capabilities you use and the extent to which you use them can vary greatly. This section is an overview of the capabilities available that can significantly affect your design and provides some reasons for why you might choose one over another. For more information and detail on any of the capabilities described, a reference to more detail is provided.

The capabilities that you need to consider include:

- Accessing the data using:
  - Embedded SQL
  - Call Level Interface (CLI)
  - Restructured Extended Executor Language (REXX)
  - Query products
- Controlling data values using:
  - Data types (built-in or user-defined)
  - Table check constraints
  - Referential integrity constraints
  - Views using the CHECK OPTION
  - Application logic and variable types
- Controlling the relationship between data values using:
  - Referential integrity constraints
  - Triggers
  - Application logic
- Executing programs at the server using:
  - Stored procedures
  - User-defined functions
  - Triggers.

You will notice that the above list mentions some capabilities such as triggers, more than once. This reflects the flexibility of these capabilities to address more than one design criteria.

The first, and most fundamental decision is whether or not to move the logic to enforce application related rules about the data into the database.

The key advantage in transferring logic focussed on the data from the application into the database is that your application becomes more independent of the data. The logic surrounding your data is centralized in one place, the database. This means that maintenance of data or data logic can be done once and affect all applications immediately.

This latter advantage is very powerful, but you must also consider that any data logic put into the database affects **all** users of the data equally. You must consider whether the rules and constraints that you wish to impose on the data apply to all users of the data or just the users of your application.

Your application requirements may also affect whether to enforce rules at the database or the application. For example, you may be required to process validation errors on data entry in a specific order. In general, these types of data validation need to be done in the application code.

You should also consider the computing environment where the application is used. You need to consider the difference between performing logic on the client machines against running the logic on the usually more powerful database server machines.

In some cases, the correct answer is to include the enforcement in both the application (perhaps due to application specific requirements) and in the database (perhaps due to other interactive uses outside the application).

Access to Data

In a relational database, you must use SQL to access the desired data, but the method with which you integrate the SQL into your application is your choice. These choices are:

- Embedded SQL
- Call Level Interface (CLI)
- REXX
- Query Products.

Embedded SQL

Embedded SQL has the advantage that it can consist of either static or dynamic SQL or a mixture of both types. If your SQL statements will be *frozen* in terms of content and format when your application is in use, you should consider using embedded static SQL in your application. With static SQL, the executor of the application can temporarily inherit the privileges of the user that bound the application to the database. Embedded dynamic SQL can be used where the statements that need to be executed are determined while the application is running. This creates a more generalized application program that can handle a greater variety of input.

An application that includes embedded SQL requires program preparation prior to using your programming language compiler. In addition, the SQL that makes up the application must be bound to the database in order for the application to run.

For additional information on using embedded SQL, refer to Chapter 2, “Writing Embedded Static Programs” on page 39.

Call Level Interface (CLI)

CLI provides data access to your application through the use of function calls to CLI functions. Any SQL statements that need to be issued are passed to the database using a CLI function call. The advantages of CLI include the elimination of the need for precompiling and binding the program, as well as the increased portability of your application through the use of the Open Database Connectivity (ODBC) interface which is supported by CLI.

An application written using CLI only uses dynamic SQL. There is some additional overhead in processing imposed by the CLI interface itself.

For additional information on CLI and ODBC, refer to “Programming With the DB2 Call Level Interface (CLI)” on page 20 and “The DB2 Call Level Interface (CLI)” on page 127.

REXX

REXX supports a dynamic SQL interface that does not require program preparation or the need for the application to be bound to the database. Using this interpretive language may help speed development of applications.

Query Products

Various query products are available that support query development and reporting. The products vary in how SQL statements are developed and the degree of logic that can be introduced. Depending on your needs, this approach may meet your requirements to access data. Further information on query products is not provided in this book.

Data Value Control

One traditional area of application logic is the validation and protection of data integrity with respect to the values allowed in the database. Applications have logic that specifically checks data values as they are entered, to ensure that the data is valid. (For example, check that the department number is a valid number and that it is for an existing department.) There are several different ways of providing these same capabilities on DB2, but from within the database.

Data Types

Every data element in the database is stored in a column of a table and the column is defined to have a data type. This data type will place certain limits on the types of values for the column. For example, an integer must be a number within a fixed range. The use of the column in SQL statements must conform to certain behaviors; for instance, an integer cannot be compared to a character string. DB2 includes a set of built-in data types with defined characteristics and behaviors. DB2 also supports defining your own data types, called *user-defined distinct types*, that are based on the built-in types but do not automatically support all the behaviors of the built-in type. You can also use data types, like binary large object (BLOB), to store data that may consist of a set of related values, such as a data structure.

For additional information on data types, refer to the *SQL Reference*.

Table Check Constraints

A table check constraint is used to define restrictions, beyond those of the data type, on the values that are allowed for a column in the table. This can be in the form of range checks or checks against other values in the same row of the same table. Table check constraints are used to enforce your rules on the data allowed in the table. If the rules apply for all applications that use the data, then using the table check constraints centralizes the rule in the database, making it generally applicable and easier to maintain.

For additional information on table check constraints, refer to CREATE TABLE in the *SQL Reference*.

Referential Integrity Constraints

Referential integrity (RI) constraints, as considered from the perspective of data value control, allow you to control the uniqueness of the values of a key (one or more columns) and existence of a row with a specified foreign key when the primary key does not exist. As with table check constraints, RI constraints are used to enforce your rules on the data, except RI constraints may span one or more tables. If the rules apply for all applications that use the data, then using the RI constraints centralizes the rules in the database, making it generally applicable and easier to maintain. See “Data Relationship Control” on page 15 for further uses of RI constraints.

For additional information on referential integrity, refer to the *SQL Reference*.

Views With Check Option

If your application has rules that cannot be defined as table check constraints or that do not apply to all uses of the data, there is another alternative to placing the rules in the application logic. You can consider creating a view of the table with the conditions on the data as part of the WHERE clause and the WITH CHECK OPTION clause specified. This view definition restricts the data that can be retrieved to the set that is valid for your application. Additionally, if the view can be updated, the WITH CHECK OPTION clause restricts updates, inserts, and deletes to the rows applicable to your application.

For additional information on the WITH CHECK OPTION, refer to the *SQL Reference*.

Application Logic and Program Variable Types

The programming language in which you write your application logic also provides some of the same restrictions on data that are described above, through declaring variables. In addition, you can choose to write code to enforce rules in the application instead of the database. Do this in cases where the rules are not generally applicable, but not in the case of views noted in “Views With Check Option.” You may also choose to place the logic in the application when you do not have control over the definitions of the data in the database, or when you believe the rule can be more effectively handled in the application logic. For example, processing errors on input data in the order that they are entered may be required, but cannot be guaranteed from the order of operations within the database.

Data Relationship Control

Another major area of focus in application logic is in the area of managing the relationships between different logical entities in your system. (For example, if a new department is added, then a new account code needs to be created.) DB2 provides two methods of managing the relationships between different objects in your database: referential integrity constraints and triggers.

Referential Integrity Constraints

Referential integrity (RI) constraints, considered from the perspective of data relationship control, allow you to control the relationships between data in more than one table. This includes rules for restricting deletes, cascading deletes, or setting null values on deletes of rows from a parent table. You can also restrict updates when the parent table does not contain a corresponding key value. RI constraints are used to enforce your rules on the data across one or more tables. If the rules apply for all applications that use the data, then using the RI constraints centralizes the rules in the database. This makes it generally applicable and easier to maintain.

For additional information on referential integrity, refer to the *SQL Reference*.

Triggers

You can use triggers in a number of different ways to support logic that can also be performed in an application. Using triggers that run before an update or insert, values that are being updated or inserted can be modified before the database is actually modified. These can be used to transform input from the application (user view of the data) to an internal database format where desired. These *before triggers* can also be used to cause other non-database operations to be activated through user-defined functions.

Using triggers that run after an update, insert or delete can be used in several ways:

- Data in the same or other tables can be updated, inserted or deleted. This is useful for maintaining relationships between data or to keeping audit trail information.
- Data can be checked against values of data in the rest of the table or in other tables. This is useful where RI constraints are not applicable or check constraints cannot be used because of references to data from other rows from this or other tables.
- Non-database operations can be activated through user-defined functions. This is useful for such things as issuing alerts or updating information outside the database.

If the rules or operations supported by the triggers apply for all applications that use the data, then using triggers centralizes the rules or operations in the database, making it generally applicable and easier to maintain.

For additional information on triggers, refer to Chapter 8, “Using the Active DBMS Capabilities” on page 349, or the *SQL Reference*.

Application Logic

You may decide to write code to enforce rules or perform related operations in the application instead of the database. You must do this for cases where the rules are not generally applicable. You may also choose to place the logic in the application when you do not have control over the definitions of the data in the database or you believe the rules or operations can be more efficiently handled in the application logic.

Application Logic at the Server

A final aspect of application design for which DB2 offers additional capability is having some of your application logic running at the database server. This is usually considered for performance reasons but can also be done for common function support.

Stored Procedures

A stored procedure is a routine for your application that is called from client application logic but runs on the database server. The most common reason to use a stored procedure is for database intensive processing that produces only small amounts of result data. This can save a large amount of communications across the network during the execution of the stored procedure. You may also consider using a stored procedure for a set of operations that are common to multiple applications. In this way, all the applications use the same logic to perform the operation.

For additional information on Stored Procedures, refer to Chapter 5, “Writing Stored Procedures” on page 167.

User-defined Functions

A user-defined function (UDF) can be written for use in performing operations within an SQL statement that returns a single scalar value. A UDF cannot contain any SQL statements. UDFs are useful for such things as transforming data values, performing calculations on one or more data values, or extracting parts of a value (such as extracting parts of a large object). They are also quite flexible, as they can be written in a high-level language (C, C++, or Java).

For additional information on writing user-defined functions, refer to Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285.

Triggers

In “Triggers” on page 15, it is noted that triggers can be used to invoke user-defined functions. This is useful when you always want a certain non-SQL operation performed when specific statements occur, or data values are changed. Examples include such operations as issuing an electronic mail message under specific circumstances or writing alert type information to a file.

For additional information on triggers, refer to Chapter 8, “Using the Active DBMS Capabilities” on page 349.

Alternatives for Coding DB2 Applications

There are three main alternatives for coding a DB2 application:

- You can use embedded SQL, which enables your application to perform any task or series of tasks that you can perform using standard SQL statements. By embedding SQL statements, your applications can work with the database, and access and manipulate data using one of the supported host programming languages. Embedded SQL is discussed in “Embedding SQL Statements in a Host Language.”
- You can use the REXX interpretive language to code your DB2 applications and immediately execute them without having to precompile, compile, or link, as you do when you use host languages. REXX is discussed in “Interactive Programming Using REXX” on page 19.
- You can use the DB2 Call Level Interface (DB2 CLI) to increase the portability of your applications by enabling independence from any one database vendor’s programming interface. DB2 CLI is discussed in “Programming With the DB2 Call Level Interface (CLI)” on page 20.

These alternatives are also discussed in “Access to Data” on page 12.

Embedding SQL Statements in a Host Language

Applications can be written with SQL statements embedded within a host language. The SQL statements provide the database interface, while the host language provides the remaining support needed for the application to execute.

Figure 2 shows an SQL statement embedded in a host language application:

Language	Example Source Code

C/C++	`strcpy( statement, "UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'");`
	`EXEC SQL EXECUTE IMMEDIATE :statement;`
	`if ( SQLCODE < 0 )`
	`   printf( "Update Error:  SQLCODE = %ld \n", SQLCODE );`

COBOL	`MOVE "UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'" TO STATEMENT`
	`EXEC SQL EXECUTE IMMEDIATE :statement END-EXEC`
	`IF SQLCODE LESS THAN 0`
	`   DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE`

FORTRAN	`statement='UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`
	`EXEC SQL EXECUTE IMMEDIATE :statement`
	`if ( sqlcode .lt. 0 ) THEN`
	`   write(*,*) 'Update error:  sqlcode = ', sqlcode`

Figure 2. Embedding SQL Statements

In this example, the `SQLCODE` field of the `SQLCA` structure is checked to determine whether the update was successful.

SQL statements placed in an application are not specific to the host language. The database manager provides a way to convert the SQL syntax to something the host language can process.

For the C, C++, COBOL or FORTRAN language, this conversion is handled by the DB2 precompiler. The precompiler converts embedded SQL statements directly into DB2 run-time services API calls. Note that compilers with integrated preprocessor support are available from non-IBM vendors. With these compilers, the conversion and compilation can be handled in one step without invoking the PREP command. The IBM precompiler is invoked using the PREP command.

When the precompiler processes a source file, it specifically looks for SQL statements and avoids the non-SQL host language. It can find SQL statements because they are surrounded by special delimiters. For the syntax information necessary to embed SQL statements in the language you are using, see the following:

- for C/C++,“Embedding SQL Statements” on page 424
- for COBOL,“Embedding SQL Statements” on page 456
- for FORTRAN,“Embedding SQL Statements” on page 479

For information on embedding SQL statements in REXX applications, see “Embedding SQL Statements” on page 490.

Figure 3 shows valid embedded SQL statements in the supported compiled host languages.

Language — **Example Source Code**

C/C++

```
 EXEC SQL
    -- SQL, or C (/* */) or C++ (//) comments allowed here
    DECLARE C1 CURSOR FOR sname;

 EXEC SQL ROLLBACK WORK
    -- SQL comments or
    /* C comments or */
    // C++ comments allowed here
 ;

 /* Only C or C++ comments allowed here */
 EXEC SQL ROLLBACK WORK;
 /* Only C or C++ comments allowed here */
```

COBOL

```
*   See COBOL documentation for comment rules.
 EXEC SQL
    -- SQL comments allowed here, and
*   full-line COBOL comments allowed here
    DECLARE C1 CURSOR FOR sname END-EXEC.

 EXEC SQL ROLLBACK WORK
*   -- SQL or full-line COBOL comments allowed here
 END-EXEC

* Only COBOL comments allowed here
 EXEC SQL ROLLBACK WORK END-EXEC
* Only COBOL comments allowed here
```

FORTRAN

```
C     Only FORTRAN comments allowed here
      EXEC SQL  -- SQL comments, and
C     full-line FORTRAN comment allowed here.
     +   ROLLBACK WORK
      I=7 ! End of line FORTRAN comments allowed here
```

Figure 3. Comments in Embedded SQL Statements

For information on the rules for embedding SQL statements into a host language or in REXX, refer to the section *Embedding SQL Statements* in one of the following chapters that corresponds to the host language you are using:

- Chapter 12, “Programming in COBOL” on page 453
- Chapter 11, “Programming in C and C++” on page 419
- Chapter 13, “Programming in FORTRAN” on page 475
- Chapter 14, “Programming in REXX” on page 489.

Interactive Programming Using REXX

REXX is an interpretive language. It contains the usual language constructs such as flow-control statements, functions, procedures, and variables that are required to build applications.

REXX applications use APIs which enable them to use most of the features provided by database manager APIs and SQL. Unlike applications written in a compiled language, REXX applications are not precompiled. Instead, all SQL statements are processed by a dynamic SQL handler. By combining REXX with these callable APIs, you have access to most of the database manager capabilities. Although some APIs supported through embedded SQL are not directly supported using REXX, they can be accessed using the DB2 Command Line Processor from within the REXX application.

As REXX is an interpretive language, you may find it is easier to develop and debug your application prototypes in REXX as compared to compiled host languages. Note that while DB2 applications coded in REXX do not provide the performance of DB2 applications that use compiled languages, they do provide the ability to create DB2 applications without precompiling, compiling, linking, or using additional software.

For details of coding and building DB2 applications using REXX, see Chapter 14, “Programming in REXX” on page 489.

Programming With the DB2 Call Level Interface (CLI)

The DB2 Call Level Interface (CLI) is IBM’s callable SQL interface to the DB2 family of database servers. It is a C and C++ application programming interface for relational database access, and it uses function calls to pass dynamic SQL statements as function arguments. A callable SQL interface is an application program interface (API) for database access, which uses function calls to invoke dynamic SQL statements. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require host variables or a precompiler.

DB2 CLI is based on the Microsoft** Open Database Connectivity (ODBC) specification, and the X/Open** specifications. These specifications were chosen as the basis for the DB2 CLI in an effort to follow industry standards, and to provide a shorter learning curve for application programmers that are familiar with either of these database interfaces.

Information regarding the ODBC support is provided in *CLI Guide and Reference*.

Creating and Preparing the Source Files

The source code is created in a standard ASCII file, called a source file, using a text editor. The source file must have the proper extension for the host language in which you write your code. See Table 23 on page 528 to find out the required file extension for the host language you are using.

Note: Not all platforms support all host languages. See your *DB2 SDK Building Applications* for specific information.

You can find the details of how to write the code for a DB2 program in “Coding a DB2 Application” on page 2, but for this discussion, assume that you have already written the source code.

If you have written your application using a compiled host language, there are additional steps required to build your application. Along with compiling and linking your program, you must *precompile* and *bind* it.

Simply stated, precompiling is the conversion of embedded SQL statements into DB2 run-time API calls that a host compiler can process, and the creation of a bind file. The bind file, which contains information on the SQL statements in the application program, is used to create a *package* in the database by using the BIND command. Optionally, the precompiler can perform the bind step at precompile time.

Binding is the process of creating a *package* from a bind file and storing it in a database. If your application accesses more than one database, a package must be created for each database.

Figure 4 on page 22 shows the order of these steps, along with the various modules of a typical compiled DB2 application. You may wish to refer to it as you read through the following sections about what happens at each stage of program preparation.

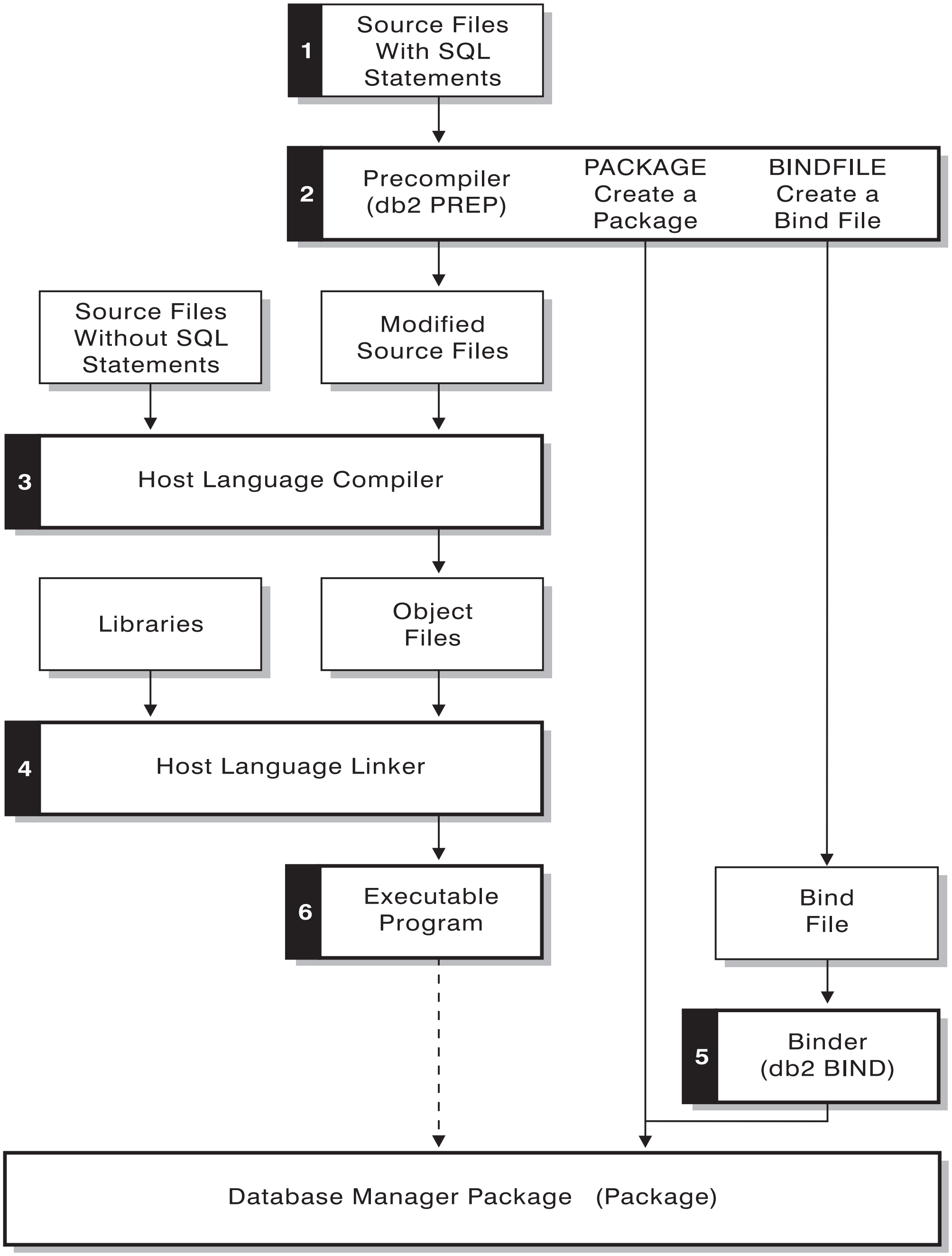


Figure 4. Preparing Programs Written in Compiled Host Languages

Creating Packages for Compiled Applications

To run applications written in compiled host languages, you must create the packages needed by the database manager at execution time. This involves the following steps:

- Precompiling, to convert embedded SQL source statements into a form the database manager can use,
- Compiling and Linking, to create the required object modules, and,
- Binding, to create the package to be used by the database manager when the program is run.

Other topics discussed in this section include:

- Application, Bind File, and Package Relationships, and,
- Rebinding, which describes when and how to rebind packages.

Since REXX is an interpreted language, applications written in this language are not precompiled, compiled, linked, or bound. Applications written in REXX connect to a database but do not bind to it. See Chapter 14, “Programming in REXX” on page 489 for more information.

Precompiling

After the source files are created, each file containing SQL statements must be precompiled with the PREP command. The precompiler converts SQL statements contained in the source file to comments, and generates the DB2 run-time API calls for those statements.

Before precompiling an application you must connect to a server, either implicitly or explicitly. Although application programs are precompiled at the client workstation and the modified source and messages are generated on the client, the server connection is needed because some of the validation is done on the server.

The precompiler also creates the information the database manager needs to process the SQL statements against a database. This information is stored in a package, in a bind file, or in both, depending on the precompiler options selected.

The precompiler generates up to four types of output:

Modified Source This file is the new version of the original source file after the precompiler converts the SQL statements into DB2 run-time API calls. It is given the appropriate host language extension.

Package If the PACKAGE option (the default) is used, or none of the BINDFILE, SYNTAX, or SQLFLAG options are specified, the package is stored in the connected database, and contains all the information required to execute the static SQL statements of a particular source file against this database only. Its name is formed from the first 8 characters of the source file name unless you specify a different name with the PACKAGE USING option.

Note: With the PACKAGE option, the database used during the precompile process must contain all of the database objects referenced by the static SQL statements in the source file. For example, a SELECT statement cannot be precompiled unless the table it references exists in the database.

Bind File If the BINDFILE option is used, the precompiler creates a bind file (with extension `.bnd`) that contains the data required to create a package. This file can be used later with the BIND command to bind the application to one or more databases. If you specify BINDFILE and do not specify the PACKAGE option, binding is deferred until you invoke the BIND command. Note that for the Command Line Processor (CLP), the default for PREP does not specify the BINDFILE option. Thus, if you are using the CLP and want the binding to be deferred, you need to specify the BINDFILE option.

If you request a bind file at precompile time but do not specify the PACKAGE, that is you do not create a package, certain object existence and authorization SQLCODES are treated as warnings instead of errors. This enables you to precompile a program and create a bind file without requiring that the referenced objects be present, or requiring that you possess the authority to execute the SQL statements being precompiled. For a list of the specific SQLCODES that are treated as warnings instead of errors see the *Command Reference*.

Message File If the MESSAGES option is used, the precompiler redirects messages to the indicated file. These messages include warnings and error messages telling of problems encountered during precompilation. If the source file does not precompile successfully, use the warning and error messages to determine the problem, correct the source file, and then attempt to precompile the source file again. If the MESSAGES option is not used, precompilation messages are written to the standard output.

A source file must always be precompiled against a specific database, even if that is not the database eventually used with the application. In practice, a test database can be used for development, and after the application is fully tested, its bind file can be bound to one or more production databases. See “Advantages of Deferred Binding” on page 28 for information about how else this can be useful.

If the code page your application uses is not the same as your database code page, you need to consider which code page to use when precompiling. See “Conversion Between Different Code Pages” on page 374.

If your application uses user-defined functions (UDFs) or user-defined distinct types (UDTs), you may need to use the FUNCPATH option when you precompile your application. This option specifies the function path that is used to resolve UDFs and UDTs for applications containing static SQL. If FUNCPATH is not specified, the default function path is "SYSIBM","SYSFUN", *"USER"*, where *"USER"* refers to the current user ID. For more information on bind options see the *Command Reference*.

To precompile an application program that accesses more than one server, you can do one of the following:

- Split the SQL statements for each database into separate source files. Do not mix SQL statements for different databases in the same file. Each source file can be precompiled against the appropriate database. This is the recommended method.
- Code your application using dynamic SQL statements only, and bind against each database your program will access.
- If all the databases look the same, that is, they have the same definition, you can group the SQL statements together into one source file.

The same procedures apply if your application will access a DRDA application server through DB2 Connect. Precompile it against the server to which it will be connecting, using the PREP options available for that server.

If you are precompiling an application that will run on DB2 for MVS/ESA or OS/390, consider using the flagger facility to check the syntax of the SQL statements. The flagger indicates SQL syntax that is supported by DB2 Universal Database, but not supported by DB2 for MVS/ESA or OS/390. You can also use the flagger to check that your SQL syntax conforms to the SQL92 Entry Level syntax. You can use the SQLFLAG option on the PREP command to invoke it and to specify the version of DB2 for MVS/ESA or OS390 SQL syntax to be used for comparison. The flagger facility will not enforce any changes in SQL use; it only issues informational and warning messages regarding syntax incompatibilities, and preprocessing is not terminated abnormally.

For details about the PREP command, see the *Command Reference*.

Compiling and Linking

Compile the modified source files and any additional source files that do not contain SQL statements using the appropriate host language compiler. The language compiler converts each modified source file into an *object module*.

See the *DB2 SDK Building Applications* book or other programming documentation for your operating platform for any exceptions to the default compile options. See your compiler's documentation for a complete description of available compile options.

The host language linker creates an executable application. For example:

- On the OS/2, Windows 3.1, Windows 95, and the Windows NT operating systems, the application can be an executable file or a dynamic link library (DLL).
- On UNIX-based systems, the application can be an executable load module or a shared library.

Note: Although applications can be DLLs on Windows platforms, for the Windows 3.1, Windows 95, or the Windows NT operating systems, the DLLs are loaded directly by the application and not by the DB2 database manager. On the

Windows 95 and Windows NT operating systems, DLLs can be loaded by the database manager. Stored procedures, which are normally built as DLLs or shared libraries are unsupported on the Windows 3.1 operating system. For information on using stored procedures, see Chapter 5, “Writing Stored Procedures” on page 167.

For information on creating executable files on other platforms supported by DB2, see the *DB2 SDK Building Applications* for those platforms.

The executable file is created by linking the following:

- User object modules, generated by the language compiler from the modified source files and other files not containing SQL statements.
- Host language library APIs, supplied with the language compiler.
- The database manager library containing the database manager APIs for your operating environment. See the *DB2 SDK Building Applications* or other programming documentation for your operating platform for the specific name of the database manager library you need for your database manager APIs.

Binding

Binding is the process that creates the package the database manager needs in order to access the database when the application is executed. Binding can be done implicitly by specifying the PACKAGE option during precompilation, or explicitly by using the BIND command against the bind file created during precompilation.

One package is created for each separately precompiled source code module. If an application has five source files, of which three require precompilation, three packages or bind files are created. By default, each package is given a name that is the same as the name of the source module from which the `.bnd` file originated, but truncated to 8 characters. If the name of this newly created package is the same as a package that currently exists in the target database, the new package replaces the previously existing package. To explicitly specify a different package name, you must use the PACKAGE USING option on the PREP command. See the *Command Reference* for details.

Renaming Packages: When creating multiple versions of an application, you should avoid conflicting names by renaming your package. For example, if you have an application called `foo` (compiled from `foo.sqc`), you precompile it and send it to all the users of your application. The users bind the application to the database, and then run the application. Subsequently, if changes are needed, you create a new version of `foo` and send this application and its bind file to the users that require the new version. The new users bind `foo.bnd` and the new application runs without any problem. However, when users attempt to run the old version of the application, they receive a timestamp conflict on the `FOO` package (which indicates that the package in the database does not match the application being run) so they rebind the client. (See “Timestamps” on page 30 for more information on package timestamps.) Now the users of the new application receive a timestamp conflict. This problem is caused because both applications use packages with the same name.

The solution is to use package renaming. When you build the first version of FOO, you precompile it with the command:

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO1
```

After you distribute this application, users can bind and run it without any problem. When you build the new version, you precompile it with the command:

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO2
```

After you distribute the new application, it will also bind and run without any problem. Since the package name for the new version is FOO2 and the package name for the first version is FOO1, there is no naming conflict and both versions of the application can be used.

Binding Dynamic Statements: For dynamically prepared statements, the statement compilation environment is determined by a number of special registers. The optimization class used is determined by the value of the CURRENT QUERY OPTIMIZATION special register. The function path used for UDF and UDT resolution is obtained from the value of the CURRENT FUNCTION PATH special register. Similarly, the value of the CURRENT EXPLAIN SNAPSHOT register determines whether explain snapshot information is captured, and the value of the CURRENT EXPLAIN MODE register determines whether explain table information is captured, for any eligible dynamic SQL statement. The default values for these special registers are the same defaults as are used for the related bind options. For information on special registers and their interaction with BIND options, refer to the appendix of the *SQL Reference*.

Resolving Unqualified Table Names: You can handle unqualified table names in your application by using one of the following methods:

- For each user, bind the package with different COLLECTION parameters from different authorization identifiers by using the following commands:

  ```
  CONNECT TO db_name USER user_name
  BIND file_name COLLECTION schema_name
  ```

  In the above example, *db_name* is the name of the database, *user_name* is the name of the user, and *file_name* is the name of the application that will be bound. Note that *db_name* and *schema_name* are usually the same value. Then use the SET CURRENT PACKAGESET statement to specify which package to use, and therefore, which qualifiers will be used. The qualifier is always the authorization identifier that is used when binding the package. For an example of how to use the SET CURRENT PACKAGESET statement, refer to the *SQL Reference*.
- Create views for each user with the same name as the table so the unqualified table names resolve correctly. (Note that the QUALIFIER option is DB2 Connect only, meaning that it can only be used when using a host server.)
- Create an alias for each user to point to the desired table.

Other Binding Considerations: If your application code page is not the same as your database code page, you may need to consider which code page to use when binding. See “Conversion Between Different Code Pages” on page 374.

If your application issues calls to any of the database manager utility APIs, such as IMPORT or EXPORT, the supplied utility bind files must be bound to the database. For details, see the *Quick Beginnings* for your platform.

You can use bind options to control certain operations that occur during binding. For example, the QUERYOPT bind option can be used to take advantage of a specific optimization class when binding, the EXPLSNAP bind option can be used to store Explain Snapshot information for eligible SQL statements in the Explain tables, and the FUNCPATH bind option can be used to properly resolve user-defined distinct types and user-defined functions in static SQL.

For information on bind options, refer to the section on the BIND command in the *Command Reference*.

If the bind process starts but never returns, it may be that other applications are connected to the database and are holding locks that you require. In this case, ensure that no applications are connected to the database. If they are, disconnect all applications on the server and the bind process will continue.

If your application will access a DRDA server, you can use the BIND options available for that server. For details on the BIND command and its options, see the *Command Reference*.

Advantages of Deferred Binding

Precompiling with binding enabled allows an application to access only the database used during the precompile process. Precompiling with binding deferred, however, allows an application to access many databases, because you can bind the BIND file against each one. This method of application development is inherently more flexible in that applications are precompiled only once, but the application can be bound to a database at any time.

Using the BIND API during execution allows an application to bind itself, perhaps as part of an installation procedure or before an associated module is executed. For example, an application can perform several tasks, only one of which requires the use of SQL statements. The application can be designed to bind itself to a database only when the task requiring SQL statements is called, and only if an associated package does not already exist.

Another advantage of the deferred binding method is that it lets you create packages without providing source code to end users. The associated bind files can be shipped with the application.

DB2 Bind File Dump Tool - db2bfd

With the DB2 Bind File Dump (db2bfd) tool, you can easily display the contents of a Version 3 bind file to examine and verify the SQL statements within it, as well as display the precompile options used to create the bind file. This may be useful in problem determination related to your application’s bind file.

The db2bfd tool is located in the misc subdirectory of the sqllib directory of the instance on AIX and UNIX-based platforms. It is located in the bin subdirectory of the sqllib directory of the instance on OS/2 and Windows platforms.

Its syntax is:

```
              ┌─────────────┐
►►──db2bfd───▼┬─-h─(1)─┬────┴─filespec─(5)──────────────────────────────►◄
              ├─-b─(2)─┤
              ├─-s─(3)─┤
              └─-v─(4)─┘
```

Notes:

1 Display the help information.

2 Display bind file header.

3 Display SQL statements.

4 Display host variable declarations

5 The name of the bind file.

Application, Bind File, and Package Relationships

A package is an object stored in the database that includes information needed to execute specific SQL statements in a single source file. A database manager application uses one package for every precompiled source file used to build the application. Each package is a separate entity, and has no relationship to any other packages used by the same or other applications. Packages are created by running the precompiler against a source file with binding enabled, or by running the binder at a later time with one or more bind files.

Database manager applications use packages for some of the same reasons that applications are compiled: improved performance and compactness. By precompiling an SQL statement, the statement is compiled into the package when the application is built, instead of at run time. Each statement is parsed, and a more efficiently interpreted operand string is stored in the package. At run time, the code generated by the precompiler calls run-time services database manager APIs with any variable information required for input or output data, and the information stored in the package is executed.

The advantages of precompilation apply only to static SQL statements. SQL statements that are executed dynamically (using PREPARE and EXECUTE or EXECUTE IMMEDIATE) are not precompiled; therefore, they must go through the entire set of processing steps at run time.

Note: Do not assume that a static version of an SQL statement automatically executes faster than the same statement processed dynamically. In some cases, this is true because of the overhead required to prepare the dynamic statement. In other cases, the same statement prepared dynamically executes faster, because the optimizer can make use of current database statistics, rather than the database statistics available at an earlier bind time. Note that if your transaction takes less than a couple of seconds to complete, static SQL will generally be faster. To choose which method to use, you should prototype both forms of binding. For a detailed comparison of static and dynamic SQL, see “Comparing Dynamic SQL with Static SQL” on page 92.

Timestamps

When generating a package or a bind file, the precompiler generates a timestamp. The timestamp is stored in the bind file or package and in the modified source file.

When an application is precompiled with binding enabled, the package and modified source file are generated with timestamps that match. When the application is run, the timestamps are checked for equality. An application and an associated package must have matching timestamps for the application to run, or an SQL0818N error is returned to the application.

Remember that when you bind an application to a database, the first eight characters of the application name are used as the package name *unless you override the default by using the PACKAGE USING option on the PREP command*. This means that if you precompile and bind two programs using the same name, the second will override the package of the first. When you run the first program, you will get a timestamp error because the timestamp for the modified source file no longer matches that of the package in the database.

When an application is precompiled with binding deferred, one or more bind files and modified source files are generated with matching timestamps. To run the application, the bind files produced by the application modules can execute. The binding process must be done for each bind file as discussed in “Binding” on page 26.

The application and package timestamps match because the bind file contains the same timestamp as the one that was stored in the modified source file during precompilation.

Rebinding

Rebinding is the process of recreating a package for an application program that was previously bound. You must rebind packages if they have been marked invalid or inoperative. In some situations, however, you may want to rebind packages that are valid. For example, you may want to take advantage of a newly created index, or make use of updated statistics after executing the RUNSTATS command.

Packages can be dependent on certain types of database objects such as tables, views, aliases, indexes, triggers, referential constraints and table check constraints. If a package is dependent on a database object (such as a table, view, trigger, and so on), and that object is dropped, the package is placed into an *invalid* state. If the object that is dropped is a UDF, the package is placed into an *inoperative* state. For more information, see the *Package Dependencies* section in the *Administration Guide*.

Invalid packages are implicitly (or automatically) rebound by the database manager when they are executed. Inoperative packages must be explicitly rebound by executing either the BIND command or the REBIND command. Note that implicit rebinding can cause unexpected errors if the implicit rebind fails. That is, the implicit rebind error is returned on the statement being executed which may not be the statement that is actually in error and if an attempt is made to execute an inoperative package, an error occurs. You may decide to explicitly rebind invalid packages rather than have the system automatically rebind them. This enables you to control when the rebinding occurs.

The choice of which command to use to explicitly rebind a package depends on the circumstances. You must use the BIND command to rebind a package for a program which has been modified to include more, fewer, or changed SQL statements. You must also use the BIND command if you need to change any bind options from the values with which the package was originally bound. In all other cases, use either the BIND or REBIND command. You should use REBIND whenever your situation does not specifically require the use of BIND, as the performance of REBIND is significantly better than that of BIND.

For details on the REBIND command, see the *Command Reference*.

Supported SQL Statements

The SQL language provides for data definition, retrieval, update, and control operations from within an application. Table 22 on page 523 shows the SQL statements supported by the DB2 product and whether the statement is supported dynamically, through the CLP, or through the CLI. You can use Table 22 on page 523 as a quick reference aid. For a complete discussion of all the statements, including their syntax, see the *SQL Reference*.

Authorization Considerations

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way. DB2 uses a set of privileges to provide protection for the information that you store in it. See the *Administration Guide* for details about the different privileges.

Most SQL statements require some type of privilege on the database objects which the statement utilizes. Most API calls usually do not require any privilege on the database objects which the call utilizes, however, many APIs require that you possess the necessary authority in order to invoke them. The DB2 APIs enable you to perform the DB2 administrative functions from within your application program. For example, to recreate a package stored in the database without the need for a bind file, you can use the `sqlarbnd` (or REBIND) API. For details on each DB2 API, refer to *API Reference*.

For information on the required privilege to issue each SQL statement, see the *SQL Reference*. For information on the required privilege and authority to issue each API call, see the *API Reference*.

An important consideration when designing your application are the privileges that users will need in order to run it. The privileges required depend on whether your application uses dynamic SQL (either embedded or CLI) or static SQL, and on which APIs it uses.

Dynamic SQL

To use dynamic SQL, the user running the application must have the privileges necessary to issue each SQL request performed, as well as the EXECUTE privilege on the package. The privileges may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

The person binding the application (for embedded dynamic SQL applications) only needs the BINDADD authority on the database, if the program contains no static SQL. Again, this privilege can be granted to the user's authorization ID, to a group of which the user is a member, or to PUBLIC.

Static SQL

To use static SQL, the user running the application only needs the EXECUTE privilege on the package. No privileges are required for each of the statements that make up the package. The EXECUTE privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

The person binding the application, however, must have the privileges necessary to perform all the statements in the application, in addition to the BINDADD authority. The privileges needed to execute the statements must be granted to the user's authorization ID or to PUBLIC. Group privileges are not used when binding static SQL statements. As with dynamic SQL, the BINDADD privilege can be granted to the user authorization ID, to a group of which the user is a member, or to PUBLIC.

The above properties of static SQL provide a way of implementing very precise control over access to information in DB2. See the example at the end of this section for a possible application of this.

Using APIs

Most of the APIs provided by DB2 do not require that any privilege be used, however, many do require some kind of authority to invoke. For the APIs that do require a privilege, the privilege must be granted to the user running the application. The privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC. For information on the required privilege and authority to issue each API call, see the *API Reference*.

Example

Consider two users, PAYROLL and BUDGET, who need to perform queries against the STAFF table. PAYROLL is responsible for paying the employees of the company, so it needs to issue a variety of SELECT statements when issuing paychecks. PAYROLL needs to be able to access each employee's salary. BUDGET is responsible for determining how much money is needed to pay the salaries. BUDGET should not, however, be able to see any particular employee's salary.

Since PAYROLL is issuing many different SELECT statements, the application you design for PAYROLL could probably make good use of dynamic SQL. This would require that PAYROLL have SELECT privilege on the STAFF table. This is not a problem since PAYROLL needs full access to the table anyhow.

BUDGET, on the other hand, should not have access to each employee's salary. This means that you should not grant SELECT privilege on the STAFF table to BUDGET. Since BUDGET does need access to the total of all the salaries in the STAFF table, you could build a static SQL application to execute a SELECT SUM(SALARY) FROM STAFF, bind the application and grant the EXECUTE privilege on your application's package to BUDGET. This lets BUDGET get the needed information without exposing the information that BUDGET should not see.

Database Manager APIs Used in Embedded or CLI Programs

Your application can use APIs to access database manager facilities that are not available using SQL statements. For complete details on the APIs available with the database manager and how to call them, see the examples in the *API Reference*.

You can use the DB2 APIs to:

- Manipulate the database manager environment, which includes cataloging and uncataloging of databases and nodes, and scanning database and node directories. You can also use APIs to create, delete, and migrate databases.
- Provide facilities to import and export data, and administer, backup, and restore the database.
- Manipulate the database manager configuration file and the database configuration files.
- Provide operations specific to the client/server environment.
- Provide the run-time interface for precompiled SQL statements. These APIs are not usually called directly by the programmer. Instead, they are inserted into the modified source file by the precompiler after processing.

Included are APIs for language vendors who want to write their own precompiler, and other APIs useful for developing applications.

For complete details on the APIs available with the database manager and how to call them, see the examples in the *API Reference*.

Setting Up the Testing Environment

In order to perform many of the tasks described in the following sections, you should set up a test environment. For example, you need a database to test your application's SQL code.

A testing environment should include the following:

- **A test database.** If your application updates, inserts, or deletes data from tables and views, test data should be used to verify its execution. If it only retrieves data from tables and views, consider using production-level data when testing it.
- **Test input data.** The input data that an application uses during testing should be valid data that represents all possible input conditions. If the application verifies that input data is valid, include both valid and invalid data to verify that the valid data is processed and the invalid data is flagged.

Creating a Test Database

If a test database must be created, create it by writing a small server application that calls the CREATE DATABASE API, or use the command line processor. See the *Command Reference* for information about the command line processor, or the *API Reference* for information about the CREATE DATABASE API.

Creating Test Tables

To design the test tables and views needed, first analyze the data needs of the application. To create a table, you need the CREATETAB authority and the CREATEIN privilege on the schema. Refer to the information on the CREATE TABLE statement in the *SQL Reference* for alternative authorities.

List the data the application accesses and describe how each data item is accessed. For example, suppose the application being developed accesses the TEST.TEMPL, TEST.TDEPT, and TEST.TPROJ tables. The type of accesses could be recorded as shown in Table 1.

Table 1. Description of the Application Data

Table or View Name	Insert Rows	Delete Rows	Column Name	Data Type	Update Access
TEST.TEMPL	No	No	EMPNO	CHAR(6)	Yes
			LASTNAME	VARCHAR(15)	Yes
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	
			JOBCODE	DECIMAL(3)	
TEST.TDEPT	No	No	DEPTNO	CHAR(3)	
			MGRNO	CHAR(6)	
TEST.TPROJ	Yes	Yes	PROJNO	CHAR(6)	Yes
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	

When the description of the application data access is complete, construct the test tables and views that are needed to test the application:

- Create a test table when the application modifies data in a table or a view. Create the following test tables using the CREATE TABLE SQL statement:
  - TEMPL
  - TPROJ

- Create a test view when the application does not modify data in the production database.

  In this example, create a test view of the TDEPT table using the CREATE VIEW SQL statement.

If the database schema is being developed along with the application, the definitions of the test tables might be refined repeatedly during the development process. Usually, the primary application cannot both create the tables and access them because the database manager cannot bind statements that refer to tables and views that do not exist. To make the process of creating and changing tables less time-consuming, consider developing a separate application to create the tables. Of course you can always create test tables interactively using the Command Line Processor (CLP).

Generating Test Data

Data can be inserted into a table using any of the following:

- INSERT...VALUES (an SQL statement) puts one or more rows into a table each time the command is issued.
- INSERT...SELECT obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement.
- The IMPORT or LOAD utility for large amounts of new or existing data.
- The RESTORE utility can be used to duplicate the contents of an existing database into an identical test database by using a BACKUP copy of the original database.

For information about the INSERT statement, see the *SQL Reference*. For information about the IMPORT, LOAD, and RESTORE utilities, see the *Administration Guide*.

A technique that you may want to use to populate your tables with randomly generated test data is embodied in the following SQL statements for a hypothetical table called EMP. Suppose this table contains four columns, ENO (employee number), LASTNAME (last name), HIREDATE (date of hire) and SALARY (employee’s salary) as in the following CREATE TABLE statement:

```
  CREATE TABLE EMP (ENO INTEGER, LASTNAME VARCHAR(30),
                    HIREDATE DATE, SALARY INTEGER);
```

Suppose you want to populate this table with employee numbers from 1 to a number, say 100, with random data for the rest of the columns. You can do this using the following SQL statement:

```
INSERT INTO EMP
-- generate 100 records
WITH DT(ENO) AS (VALUES(1) UNION ALL SELECT ENO+1 FROM DT WHERE ENO < 100 )

-- Now, use the generated records in DT to create other columns
-- of the employee record.
  SELECT ENO,
    TRANSLATE(CHAR(INTEGER(RAND()*1000000)),
              CASE MOD(ENO,4) WHEN 0 THEN 'aeiou' || 'bcdfg'
                              WHEN 1 THEN 'aeiou' || 'hjklm'
                              WHEN 2 THEN 'aeiou' || 'npqrs'
                                     ELSE 'aeiou' || 'twxyz' END,
                                          '1234567890') AS LASTNAME,
    CURRENT DATE - (RAND()*10957) DAYS AS HIREDATE,
    INTEGER(10000+RAND()*200000) AS SALARY
  FROM DT;

SELECT * FROM EMP;
```

Figure 5. Generating Test Data

The following is an explanation of the above statement:

1. The first part of the above INSERT statement generates 100 records for the first 100 employees using a recursive subquery to generate the employee numbers. Each record contains the employee number. To change the number of employees, use a number other than 100.
2. The SELECT statement generates the LASTNAME column. It begins by generating a random integer up to 6 digits long using the RAND function. It then converts the integer to its numeric character format using the CHAR function.
3. To convert the numeric characters to alphabet characters, the statement uses the TRANSLATE function to convert the ten numeric characters (0 through 9) to alphabet characters. Since there are more than 10 alphabet characters, the statement selects from five different translations. This results in names having enough random vowels to be pronounceable and so the vowels are included in each translation.
4. The statement generates a random HIREDATE value. The value of HIREDATE ranges back from the current date to 30 years ago. HIREDATE is calculated by subtracting a random number of days between 0 and 10 957 from the current date. (10 957 is the number of days in 30 years.)
5. Finally, the statement randomly generates the SALARY. The minimum salary is 10 000, to which a random number from 0 to 200 000 is added.

For example programs that are helpful in generating random test date, please see the `fillcli.sqc` and `fillsrv.sqc`sample programs in the `sqllib/samples/c` subdirectory.

You may also want to consider prototyping any user-defined functions (UDF) you are developing against the test data. For more information on why and how you write

UDFs, see Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285 and “User-Defined Functions (UDF)” on page 254.

Running, Testing and Debugging Your Programs

The *DB2 SDK Building Applications* book for your operating platform tells you how to run your program in your environment. You can do the following to help you during the testing and debugging of your code:

- Use the same techniques discussed in “Prototyping Your SQL Statements.” These include using the command line processor, the Explain facility, analyzing the system catalog views for the information about the tables and databases your program is manipulating, and updating certain system catalog statistics to simulate production conditions.
- Use the database system monitor to capture certain optimizing information for analysis. See the *System Monitor Guide and Reference*.
- Use the flagger facility to check the syntax of SQL statements in applications being developed for DB2 for MVS/ESA or OS/390, or for conformance to the SQL92 Entry Level standard. This facility is invoked during precompilation. For information about how to do this, see “Precompiling” on page 23, towards the end of the section.
- Make full use of the error-handling APIs. For instance, print all the messages during the testing phase. See the *API Reference* for information about APIs.

Prototyping Your SQL Statements

As you design and code your application, you can take advantage of certain database manager features and tools to prototype portions of your SQL code, and to improve performance. For example, you can do the following:

- Use the command line processor (CLP) to test many SQL statements before having to compile and link a complete program.

  This allows you to define and manipulate information stored in a database table, index, or view. You can add, delete, or update information as well as generate reports from the contents of tables. Note that you have to minimally change the CLP syntax for some SQL statements in order to use host variables in your program. Host variables are used to store data that is output to your screen using the CLP. In addition, some embedded SQL statements (such as BEGIN DECLARE SECTION) are not supported by the CLP as they are not relevant to that environment. See Table 22 on page 523 to see which SQL statements are not supported by the CLP.

  You can also redirect the input and output of command line processor requests. For example, you could create one or more files containing SQL statements you need as input into a command line processor request, thereby not having to keep retyping the statement.

  For information about the command line processor, see the *Command Reference*.

- Use the Explain facility to get an idea of the estimated costs of the DELETE, INSERT, UPDATE, or SELECT statements you plan to use in your program. The Explain facility places the information about the structure and the estimated costs of the subject statement into user supplied tables. You can view this information using Visual Explain or the db2exfmt utility.

  For information about how to use the Explain facility, see the *Administration Guide*.

- Use the system catalog views to easily retrieve information about existing databases. The system catalog tables on which the views are based are created and maintained by the database manager during normal operation as databases are created, altered, and updated. These views contain data about each database, including such things as authorities granted, column names, data types, indexes, package dependencies, referential constraints, table names, views, and so on. Data in the system catalog views is available through normal SQL query facilities.

  You can update some system catalog views containing statistical information used by the SQL optimizer. Some columns in these views may be changed to influence the optimizer or to investigate the performance of hypothetical databases. You can use this method to simulate a production system on your development or test system and analyze how queries perform.

  For a complete description of each system catalog view, see the appendix in the *SQL Reference*. For information about system catalog statistics and which ones you can change, see the *Administration Guide*.

Chapter 2. Writing Embedded Static Programs

This chapter describes the characteristics of and reasons for using embedded static SQL in your applications. It discusses the following topics:

- Characteristics and Reasons for Using Static SQL
- Selecting Multiple Rows Using a Cursor
- Updating and Deleting Retrieved Data
- Advanced Scrolling Techniques
- Diagnostic Handling and the SQLCA Structure

Characteristics and Reasons for Using Static SQL

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as *static* SQL. This is in contrast to *dynamic* SQL statements whose syntax is not known until run time.

Note: Static SQL is not supported in interpreted languages, such as REXX.

The structure of an SQL statement must be completely specified in order for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at run time are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled.

When a static SQL statement is prepared, an executable form of the statement is created and stored in the package in the database. The executable form can be constructed either at precompile time, or at a later bind time. In either case, preparation occurs *before* run time. The authorization of the person binding the application is used, and optimization is based upon database statistics and configuration parameters that may not be current when the application runs.

Advantages of Static SQL

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time services API calls that the host language compiler can process.

Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or a range of values.

Static SQL statements are *persistent*, meaning that the statements last for as long as the package exists. Dynamic SQL statements are cached until they are either invalidated, freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2 SQL compiler whenever a cached statement becomes invalid. For information on caching and the reasons for invalidation of a cached statement, see the *Notes* section for the EXECUTE statement in the *SQL Reference*.

The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at run time, while dynamic SQL must be explicitly compiled at run time (for example, by using the PREPARE statement). Because DB2 caches dynamic SQL statements, the statements do not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.

There can be performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically since the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

Note: The performance of static SQL depends on the statistics of the database the last time the application was bound. However, if these statistics change, the performance of equivalent dynamic SQL can be very different. If, for example, an index is added to a database at a later time, an application using static SQL cannot take advantage of the index unless it is re-bound to the database. In addition, if you are using host variables in a static SQL statement, the optimizer will not be able to take advantage of any distribution statistics for the table.

Example Static SQL Program

This sample program shows examples of static SQL statements and database manager API calls in supported languages. (The REXX language does not support static SQL, so a sample is not provided.)

This sample program contains a query that selects a single row. Such a query can be performed using the SELECT INTO statement.

The SELECT INTO statement selects one row of data from tables in a database, and the values in this row are assigned to host variables specified in the statement. Host variables are discussed in detail in “Using Host Variables” on page 46. For example, the following statement will deliver the salary of the employee with the last name of 'HAAS' into the host variable empsal:

```
SELECT SALARY
  INTO :empsal
  FROM EMPLOYEE
  WHERE LASTNAME='HAAS'
```

A SELECT INTO statement must be specified to return only one or zero rows. Finding more than one row results in an error, SQLCODE -811 (SQLSTATE 21000). If several rows can be the result of a query, a cursor must be used to process the rows. See “Selecting Multiple Rows Using a Cursor” on page 55 for more information.

For more details on the SELECT INTO statement, refer to the *SQL Reference*.

For an introductory discussion on how to write SELECT statements, see “Coding SQL Statements to Retrieve and Manipulate Data” on page 46.

How the Example Static SQL Program Works

1. **Include the SQLCA.** The INCLUDE SQLCA statement defines and declares the SQLCA structure, and defines SQLCODE and SQLSTATE as elements within the structure. The SQLCODE field of the SQLCA structure is updated with diagnostic information by the database manager after every execution of SQL statements or database manager API calls.

2. **Declare host variables.** The SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. These are variables that can be referenced in SQL statements. Host variables are used to pass data to the database manager or to hold data returned by it. They are prefixed with a colon (:) when referenced in an SQL statement. For more information, see “Using Host Variables” on page 46.

3. **Connect to database.** The program connects to the sample database, and requests shared access to it. (It is assumed that a START DATABASE MANAGER API call or db2start command has been issued.) Other programs that connect to the same database using shared access are also granted access.

4. **Retrieve data.** The SELECT INTO statement retrieves a single value based upon a query. This example retrieves the FIRSTNME column from the EMPLOYEE table where the value of the LASTNAME column is JOHNSON. The value SYBIL is returned and placed in the host variable firstname. The sample tables supplied with DB2 are listed in the appendix of the *SQL Reference*.

5. **Process errors.** For C, the macro CHECKERR (which calls the procedure check_error) is called to query the state of the SQL statement. This procedure is located in the util.c program and is included when this example is compiled. For COBOL, the error checking utility is in the checkerr.cbl file. For FORTRAN, the error checking utility is in the util.f file.

   For more information on how to include the error checking utility see the README file in the programming language specific samples directory, and see “Using GET ERROR MESSAGE in Example Programs” on page 85.

6. **Disconnect from database.** The program disconnects from the database by executing the CONNECT RESET statement.

C Example: STATIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;  1

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;  2
      char firstname[13];
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: STATIC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;  3
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: static [userid passwd]\n\n");
      return 1;
   } /* endif */

   EXEC SQL SELECT FIRSTNME INTO :firstname   4
            FROM employee
            WHERE LASTNAME = 'JOHNSON';
   CHECKERR ("SELECT statement");  5

   printf( "First name = %s\n", firstname );

   EXEC SQL CONNECT RESET;  6
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : STATIC.SQC */
```

COBOL Example: STATIC.SQB

```
 Identification Division.
 Program-ID. "static".

 Data Division.
 Working-Storage Section.
     copy "sql.cbl".
     copy "sqlca.cbl".  1

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  2
 01 firstname         pic x(12).
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: STATIC".

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd  3
        END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL SELECT FIRSTNME INTO :firstname  4
           FROM EMPLOYEE
           WHERE LASTNAME = 'JOHNSON' END-EXEC.
     move "SELECT" to errloc.
     call "checkerr" using SQLCA errloc.                          5

     display "First name = ", firstname.

     EXEC SQL CONNECT RESET END-EXEC.                             6
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.

 End-Prog.
     stop run.
```

FORTRAN UNIX Example: STATIC.SQF

```
      program static
      implicit none

      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA  1

      EXEC SQL BEGIN DECLARE SECTION  2
        character*12   firstname
        character*8    userid
        character*18   passwd
      EXEC SQL END DECLARE SECTION

      character*80     errloc

      print *, 'Sample Fortran Program: STATIC'

      print *, 'Enter your user id (default none):'
      read 100, userid
100   format (a8)

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO sample
      else
      print *, 'Enter your password :'
      read 100, passwd

      EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if  3
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL SELECT firstnme INTO :firstname
     c FROM employee WHERE lastname = 'JOHNSON'  4
      errloc = 'SELECT'
      call checkerr (sqlca, errloc, *999)  5

      print *, 'First name = ', firstname

      EXEC SQL CONNECT RESET  6
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

  999 stop
      end
```

Coding SQL Statements to Retrieve and Manipulate Data

The database manager provides application programmers with statements for retrieving and manipulating data; the coding task consists of embedding these statements into the host language code. This section shows how to code statements that will retrieve and manipulate data for one or more rows of data in DB2 tables. (It does not go into the details of the different host languages.) For the exact rules of placement, continuation, and delimiting SQL statements, see:

- Chapter 12, “Programming in COBOL” on page 453
- Chapter 11, “Programming in C and C++” on page 419
- Chapter 13, “Programming in FORTRAN” on page 475
- Chapter 14, “Programming in REXX” on page 489.

Retrieving Data

One of the most common tasks of an SQL application program is to retrieve data. This is done using the *select-statement*, which is a form of query that searches for rows of tables in the database that meet specified search conditions. If such rows exist, the data is retrieved and put into specified variables in the host program, where it can be used for whatever it was designed to do.

After you have written a *select-statement*, you code the SQL statements that define how information will be passed to your application.

You can think of the result of a select-statement as being a table having rows and columns, much like a table in the database. If only one row is returned, you can deliver the results directly into host variables specified by the SELECT INTO statement.

If more than one row is returned, you must use a *cursor* to fetch them one at a time. A cursor is a named control structure used by an application program to point to a specific row within an ordered set of rows. For information about how to code and use cursors, see the following sections:

- “Declaring and Using the Cursor” on page 55,
- “Selecting Multiple Rows Using a Cursor” on page 55,
- “Example Cursor Program” on page 58.

Using Host Variables

Host variables are variables referenced by embedded SQL statements. They transmit data between the database manager and an application program. When you use a host variable in an *SQL statement*, you must prefix its name with a colon, (**:**). When you use a host variable in a *host language statement*, omit the colon.

Host variables are declared in compiled host languages, and are delimited by BEGIN DECLARE SECTION and END DECLARE SECTION statements. These statements enable the precompiler to find the declarations.

Host variables are declared using a subset of the host language. For a description of the supported syntax for your host language, see:

- Chapter 12, “Programming in COBOL” on page 453

- Chapter 11, “Programming in C and C++” on page 419
- Chapter 13, “Programming in FORTRAN” on page 475
- Chapter 14, “Programming in REXX” on page 489.

The following rules apply to host variable declaration sections:

- All host variables must be declared in the source file before they are referenced, except for host variables referring to SQLDA structures.
- Multiple declare sections may be used in one source file.
- The precompiler is unaware of host language variable scoping rules.

  With respect to SQL statements, all host variables have a global scope regardless of where they are actually declared in a single source file. Therefore, host variable names must be unique within a source file.

  This does not mean that the DB2 precompiler changes the scope of host variables to global so that they can be accessed outside the scope in which they are defined. Consider the following example:

  ```
  foo1(){
    .
    .
    .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
  x=10;
    .
    .
    .
  }


  foo2(){
    .
    .
    .
    y=x;
    .
    .
    .
  }
  ```

  Depending on the language, the above example will either fail to compile because variable x is not declared in function foo2() or the value of x would not not be set to 10 in foo2(). To avoid this problem, you must either declare x as a global variable, or pass x as a parameter to function foo2() as follows:

```
foo1(){
.
.
.
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
  x=10;
  foo2(x);
.
.
.
}


foo2(){
.
.
.
  y=x;
.
.
.
}
```

Figure 6 shows how to declare host variables, while Figure 7 shows how host variables are referenced in the supported host languages. For information on naming and referencing host variables in REXX, see “Naming Host Variables” on page 492 and “Referencing Host Variables” on page 492.

Language	Example Source Code

C/C++	EXEC SQL BEGIN DECLARE SECTION;
	  short dept=38, age=26;
	  double salary;
	  char CH;
	  char name1[9], NAME2[9];
	  /* C comment */
	  short nul_ind;
	EXEC SQL END DECLARE SECTION;

COBOL	EXEC SQL BEGIN DECLARE SECTION END-EXEC.
	   01 age PIC S9(4) COMP-5 VALUE 26.
	   01 DEPT PIC S9(9) COMP-5 VALUE 38.
	   01 salary PIC S9(6)V9(3) COMP-3.
	   01 CH PIC X(1).
	   01 name1 PIC X(8).
	   01 NAME2 PIC X(8).
	* COBOL comment
	   01 nul-ind PIC S9(4) COMP-5.
	 EXEC SQL END DECLARE SECTION END-EXEC.

FORTRAN	EXEC SQL BEGIN DECLARE SECTION
	         integer*2 age /26/
	         integer*4 dept /38/
	         real*8 salary
	         character ch
	         character*8 name1,NAME2
	C FORTRAN comment
	         integer*2 nul_ind
	       EXEC SQL END DECLARE SECTION

Figure 6. Declaring Host Variables

Language	Example Source Code

C/C++	EXEC SQL FETCH C1 INTO :cm;
	printf("Commission = %f\n", cm);

COBOL	EXEC SQL FETCH C1 INTO :cm END-EXEC DISPLAY 'Commission = ' cm
FORTRAN	EXEC SQL FETCH C1 INTO :cm WRITE(*,*) 'Commission = ', cm

Figure 7. Referencing Host Variables

Using Indicator Variables

Applications must prepare for receiving null values by associating an *indicator variable* with any host variable that can receive a null. An indicator variable is shared by both the database manager and the host application; therefore, the indicator variable must be declared in the application as a host variable. This host variable corresponds to the database manager type SMALLINT.

An indicator variable is placed in an SQL statement immediately after the host variable, and is prefixed with a colon. A space can separate the indicator variable from the host variable, but is not required. However, do not put a comma between the host variable and the indicator variable. You can also specify an indicator variable by using the optional INDICATOR keyword, which you place between the host variable and its indicator.

Figure 8 shows indicator variable usage in the supported host languages using the INDICATOR keyword.

Language	Example Source Code

C/C++	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;`
	`if ( cmind < 0 )`
	`   printf( "Commission is NULL\n" );`

COBOL	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC`
	`IF cmind LESS THAN 0`
	`   DISPLAY 'Commission is NULL'`

FORTRAN	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind`
	`IF ( cmind .LT. 0 ) THEN`
	`   WRITE(*,*) 'Commission is NULL'`
	`ENDIF`

Figure 8. Indicator Variables

In the figure, `cmind` is examined for a negative value. If it is not negative, the application can use the returned value of `cm`. If it is negative, the fetched value is NULL and `cm` should not be used[3]. The database manager does not change the value of the host variable in this case.

If the data type can handle NULLs, the application must provide a NULL indicator. Otherwise, an error may occur. If a NULL indicator is not used, an SQLCODE -305 (SQLSTATE 22002) is returned.

If the SQLCA structure indicates a truncation warning, the indicator variables can be examined for truncation. If an indicator variable has a positive value, a truncation occurred.

- If the seconds portion of a TIME data type is truncated, the indicator value contains the seconds portion of the truncated data.
- For all other string data types, except large objects (LOB), the indicator value represents the actual length of the data returned. User-defined distinct types (UDT) are handled in the same way as their base type.

[3] If the database configuration parameter DFT_SQLMATWARN is set to 'YES', the value of `cmind` may be -2. This indicates a NULL that was caused by evaluating an expression with an arithmetic error or by an overflow while attempting to convert the numeric result value to the host variable.

When processing INSERT or UPDATE statements, the database manager checks the indicator variable if one exists. If the indicator variable is negative, the database manager sets the target column value to NULL if NULLs are allowed. If the indicator variable is zero or positive, the database manager uses the value of the associated host variable.

The SQLWARN1 field in the SQLCA structure may contain an 'X' or 'W' if the value of a string column is truncated when it is assigned to a host variable. It contains an 'N' if a null terminator is truncated.

A value of 'X' is returned by the database manager only if all of the following conditions are met:

- A mixed code page connection exists where conversion of character string data from the database code page to the application code page involves a change in the length of the data.
- A cursor is blocked.
- An indicator variable is provided by your application.

The value returned in the indicator variable will be the length of the resultant character string in the application's code page.

In all other cases involving data truncation, (as opposed to NULL terminator truncation), the database manager returns a 'W'. In this case, the database manager returns a value in the indicator variable to the application that is the length of the resultant character string in the code page of the select list item (either the application code page, the data base code page, or nothing). For related information, see the table, *Fields of SQLCA*, in the *SQL Reference*.

Data Types

Each column of every DB2 table is given an *SQL data type* when the column is created. For information about how these types are assigned to columns, see the CREATE TABLE statement in the *SQL Reference*. The database manager supports the following column data types:

SMALLINT 16-bit signed integer

INTEGER 32-bit signed integer. **INT** can be used as a synonym for this type.

DOUBLE Double-precision floating point. **DOUBLE PRECISION** and **FLOAT(n)** (where n is greater than 24) are synonyms for this type.

REAL Single-precision floating point. **FLOAT(n)** (where n is less than 24) is a synonym for this type.

DECIMAL Packed decimal. **DEC**, **NUMERIC**, and **NUM** are synonyms for this type.

CHAR Fixed-length character string of length 1 byte to 254 bytes. **CHARACTER** can be used as a synonym for this type.

VARCHAR Variable-length character string of length 1 byte to 4000 bytes. **CHARACTER VARYING** and **CHAR VARYING** are synonyms for this type.

LONG VARCHAR Long variable-length character string of length 1 byte to 32 700 bytes.

CLOB Large object variable-length character string of length 1 byte to 2 gigabytes.

BLOB Large object variable-length binary string of length 1 byte to 2 gigabytes.

DATE Character string of length 10 representing a date.

TIME Character string of length 8 representing a time.

TIMESTAMP Character string of length 26 representing a timestamp.

The following data types are supported only in double-byte character set (DBCS) and Extended UNIX Code (EUC) character set environments:

GRAPHIC Fixed-length graphic string of length 1 to 127 double-byte characters.

VARGRAPHIC Variable-length graphic string of length 1 to 2000 double-byte characters.

LONG VARGRAPHIC Long variable-length graphic string of length 1 to 16 350 double-byte characters.

DBCLOB Large object variable-length graphic string of length 1 to 1 073 741 823 double-byte characters.

Notes:

1. Every supported data type can have the NOT NULL attribute. This is treated as another type.
2. The above set of data types can be extended by defining user-defined distinct types (UDT). UDTs are separate data types which use the representation of one of the built-in SQL types.

Supported host languages have data types that correspond to the majority of the database manager data types. Only these host language data types can be used in host variable declarations. When the precompiler finds a host variable declaration, it determines the appropriate SQL data type value. The database manager uses this value to convert the data exchanged between itself and the application.

As the application programmer, it is important for you to understand how the database manager handles comparisons and assignments between different data types. Simply put, data types must be compatible with each other during assignment and comparison operations, whether the database manager is working with two SQL column data types, two host-language data types, or one of each.

The *general* rule for data type compatibility is that all supported host-language numeric data types are comparable and assignable with all database manager numeric data types, and all host-language character types are compatible with all database manager character types; numeric types are incompatible with character types. However, there are also some exceptions to this general rule depending on host language idiosyncrasies and limitations imposed when working with large objects.

Within SQL statements, DB2 provides conversions between compatible data types. For example, in the following SELECT statement, SALARY and BONUS are DECIMAL columns; however, each employee's total compensation is returned as DOUBLE data:

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

Note that the execution of the above statement includes conversion between DECIMAL and DOUBLE data types. To make the query results more readable on your screen, you could use the following SELECT statement:

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

To convert data within your application, contact your compiler vendor for additional routines, classes, built-in types, or APIs that supports this conversion.

Character data types may also be subject to character conversion. If your application code page is not the same as your database code page, see "Conversion Between Different Code Pages" on page 374.

For the list of supported SQL data types and the corresponding host language data types, see the following:

- for C/C++, "Supported SQL Data Types" on page 445
- for COBOL,"Supported SQL Data Types" on page 468

- for FORTRAN,“Supported SQL Data Types” on page 485
- for REXX,“Supported SQL Data Types” on page 497 .

See the table on *Data Type Compatibility for Assignments and Comparisons* in the *SQL Reference* for more details about compatible data types. For more information about SQL data types, the rules of assignments and comparisons, and data conversion and conversion errors, see the *SQL Reference*.

Using an Indicator Variable in the STATIC program

The following code segments show the modification to the corresponding segments in the C version of the sample STATIC program, listed in “C Example: STATIC.SQC” on page 43. They show the implementation of indicator variables on data columns that are nullable. In this example. the STATIC program is extended to select another column, WORKDEPT. This column can have a null value. An indicator variable needs to be declared as a host variable before being used.

```
  .
  .
  .
EXEC SQL BEGIN DECLARE SECTION;
  char  wd[3];
  short wd_ind;
  char firstname[13];
  .
  .
  .

EXEC SQL END DECLARE SECTION;
  .
  .
  .
/* CONNECT TO SAMPLE DATABASE */
  .
  .
  .
EXEC SQL SELECT FIRSTNME, WORKDEPT INTO :firstname, :wd:wdind
  FROM EMPLOYEE
  WHERE LASTNAME = 'JOHNSON';
  .
  .
  .
```

Selecting Multiple Rows Using a Cursor

To allow an application to retrieve a set of rows, SQL uses a mechanism called a *cursor*.

To help understand the concept of a cursor, assume that the database manager builds a *result table* to hold all the rows retrieved by executing a SELECT statement. A cursor makes rows from the result table available to an application, by identifying or pointing to a *current row* of this table. When a cursor is used, an application can retrieve each row sequentially from the result table until an end of data condition, that is, the NOT FOUND condition, SQLCODE +100 (SQLSTATE 02000) is reached. The set of rows obtained as a result of executing the SELECT statement can consist of zero, one, or more rows, depending on the number of rows that satisfy the search condition.

The steps involved in processing a cursor are as follows:

1. Specify the cursor using a DECLARE CURSOR statement.
2. Perform the query and build the result table using the OPEN statement.
3. Retrieve rows one at a time using the FETCH statement.
4. Process rows with the DELETE or UPDATE statements (if required).
5. Terminate the cursor using the CLOSE statement.

An application can use several cursors concurrently. Each cursor requires its own set of DECLARE CURSOR, OPEN, CLOSE, and FETCH statements.

See “Example Cursor Program” on page 58 for an example of how an application can select a set of rows and, using a cursor, process the set one row at a time.

Declaring and Using the Cursor

The DECLARE CURSOR statement defines and names the cursor, identifying the set of rows to be retrieved using a SELECT statement.

The application assigns a name for the cursor. This name is referred to in subsequent OPEN, FETCH, and CLOSE statements. The query is any valid select statement.

Figure 9 shows a DECLARE statement associated with a static SELECT statement.

Language	Example Source Code

C/C++	EXEC SQL DECLARE C1 CURSOR FOR
	  SELECT PNAME, DEPT FROM STAFF
	  WHERE JOB=:*host_var*;

COBOL	EXEC SQL DECLARE C1 CURSOR FOR
	  SELECT NAME, DEPT FROM STAFF
	    WHERE JOB=:*host-var* END-EXEC.

FORTRAN	EXEC SQL DECLARE C1 CURSOR FOR + SELECT NAME, DEPT FROM STAFF + WHERE JOB=:*host_var*

Figure 9. Declare Cursor Statement

Note: The placement of the DECLARE statement is arbitrary, but it must be placed above the first use of the cursor.

Cursors and Unit of Work Considerations

The actions of a COMMIT or ROLLBACK operation vary for cursors, depending on how the cursors are declared.

Read Only Cursors

If a cursor is determined to be read only and uses a repeatable read isolation level, repeatable read locks are still gathered and maintained on system tables needed by the unit of work. Therefore, it is important for applications to periodically issue COMMIT statements, even for read only cursors. For information on the repeatable read isolation level, see “Repeatable Read” on page 134.

WITH HOLD Option

If an application completes a unit of work by issuing a COMMIT statement, *all open cursors*, except those declared using the WITH HOLD option, are automatically closed by the database manager.

A cursor that is declared WITH HOLD maintains the resources it accesses across multiple units of work. The exact effect of declaring a cursor WITH HOLD depends on how the unit of work ends.

If the unit of work ends with a COMMIT statement, open cursors defined WITH HOLD remain OPEN. The cursor is positioned before the next logical row of the result table. In addition, prepared statements referencing OPEN cursors defined WITH HOLD are retained. Only FETCH and CLOSE requests associated with a particular cursor are valid immediately following the COMMIT. UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF statements are valid only for rows fetched within the same unit of work. If a package is rebound during a unit of work, all held cursors are closed.

If the unit of work ends with a ROLLBACK statement, all open cursors are closed, all locks acquired during the unit of work are released, and all prepared statements that are dependent on work done in that unit are dropped.

For example, suppose that the TEMPL table contains 1000 entries. You want to update the salary column for all employees, and you expect to issue a COMMIT statement every time you update 100 rows.

1. Declare the cursor using the WITH HOLD option:

   ```
   EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
     SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
     FROM TEMPL FOR UPDATE OF SALARY
   ```

2. Open the cursor and fetch data from the result table one row at a time:

   ```
   EXEC SQL OPEN EMPLUPDT
     .
     .
     .

   EXEC SQL FETCH EMPLUPDT
     INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
   ```

3. When you want to update or delete a row, use an UPDATE or DELETE statement using the WHERE CURRENT OF option. For example, to update the current row, your program can issue:

   ```
   EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
     WHERE CURRENT OF EMPLUPDT
   ```

4. After a COMMIT is issued, you must issue a FETCH before you can update another row.

You should include code in your application to detect and handle an SQLCODE -501 (SQLSTATE 24501), which can be returned on a FETCH or CLOSE statement if your application either:

- Uses cursors declared WITH HOLD
- Executes more than one unit of work and leaves a WITH HOLD cursor open across the unit of work boundary (COMMIT WORK).

If an application invalidates its package by dropping a table on which it is dependent, the package gets rebound dynamically. If this is the case, an SQLCODE -501 (SQLSTATE 24501) is returned for a FETCH or CLOSE statement because the database manager closes the cursor. The way to handle an SQLCODE -501 (SQLSTATE 24501) in this situation depends on whether you want to fetch rows from the cursor.

- If you want to fetch rows from the cursor, open the cursor, then run the FETCH statement. Note, however, that the OPEN statement repositions the cursor to the start. The previous position held at the COMMIT WORK statement is lost.
- If you do not want to fetch rows from the cursor, do not issue any more SQL requests against the cursor.

Example Cursor Program

This sample program shows the SQL statements that define and use a cursor. The cursor is processed using static SQL.

Since REXX does not support static SQL, a sample is not provided. See "Example Dynamic SQL Program" on page 96 for a REXX example that processes a cursor dynamically.

How the Example Cursor Program Works

1. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor `c1` to a query. The query identifies the rows that the application retrieves using the FETCH statement. The `job` field of `staff` is defined to be updatable, even though it is not specified in the result table.
2. **Open the cursor.** The cursor `c1` is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
3. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *current* row.
4. **Close the cursor.** The CLOSE statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C `check_error` is redefined as `CHECKERR` and is located in the `util.c` file.

COBOL `CHECKERR` is an external program named `checkerr.cbl`

FORTRAN `CHECKERR` is a subroutine located in the `util.f` file.

See "Using GET ERROR MESSAGE in Example Programs" on page 85 for the source code for this error checking utility.

C Example: CURSOR.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: CURSOR \n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
         CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: cursor [userid passwd]\n\n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR  1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;  2
   CHECKERR ("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :pname, :dept;  3
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
         pname, dept );
   } while ( 1 );

   EXEC SQL CLOSE c1;  4
   CHECKERR ("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   CHECKERR ("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : CURSOR.SQC */
```

COBOL Example: CURSOR.SQB

```
 Identification Division.
 Program-ID. "cursor".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 pname             pic x(10).
 77 dept              pic s9(4) comp-5.
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: CURSOR".

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
        END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR                                  1
              SELECT name, dept FROM staff
              WHERE job='Mgr'
              FOR UPDATE OF job END-EXEC.

     EXEC SQL OPEN c1 END-EXEC.                                      2
     move "OPEN CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

     EXEC SQL CLOSE c1 END-EXEC.                                     4
```

```
    move "CLOSE CURSOR" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL ROLLBACK END-EXEC.
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.
    DISPLAY "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :PNAME, :DEPT END-EXEC.  3
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

FORTRAN UNIX Example: CURSOR.SQF

```
      program cursor
      implicit none

      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA
      EXEC SQL BEGIN DECLARE SECTION
        character*10  pname
        integer*2     dept
        character*8   userid
        character*18  passwd
      EXEC SQL END DECLARE SECTION

      character*80    errloc

      print *, 'Sample FORTRAN program: CURSOR'

      print *, 'Enter your user id (default none):'
      read 101, userid
101   format (a8)

      if( userid(1:1) .eq. ' ' ) then
       EXEC SQL CONNECT TO sample
      else
       print *, 'Enter your password :'
       read 101, passwd

       EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL DECLARE c1 CURSOR FOR  1
     c         SELECT name, dept FROM staff WHERE job='Mgr'
     c         FOR UPDATE OF job

      EXEC SQL OPEN c1  2
      errloc = 'OPEN'
      call checkerr (sqlca, errloc, *999)

  100 continue
      EXEC SQL FETCH c1 INTO :pname, :dept  3
      if (sqlcode .ne. 0) goto 200
      print *, pname, ' in dept. ', dept, ' will be demoted to Clerk.'
      goto 100

  200 EXEC SQL CLOSE c1  4
      errloc = 'CLOSE'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL ROLLBACK
      errloc = 'ROLLBACK'
      call checkerr (sqlca, errloc, *999)
      print *, 'On second thought -- changes rolled back.'

      EXEC SQL CONNECT RESET
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

  999 stop
      end
```

Updating and Deleting Retrieved Data

It is possible to update and delete the row referenced by a cursor. For a row to be updatable, the query corresponding to the cursor must not be read-only. (For a description of what makes a query updatable or deletable, see the *SQL Reference*.)

Updating Retrieved Data

To update with a cursor, use the WHERE CURRENT OF clause in an UPDATE statement. Use the FOR UPDATE clause to tell the system that you want to update some columns of the result table. You can specify a column in the FOR UPDATE without it being in the fullselect; therefore, you can update columns that are not explicitly retrieved by the cursor. If the FOR UPDATE clause is specified without column names, all columns of the table or view identified in the first FROM clause of the outer fullselect are considered to be updatable. Do not name more columns than you need in the FOR UPDATE clause. In some cases, naming extra columns in the FOR UPDATE clause can cause DB2 to be less efficient in accessing the data.

Deleting Retrieved Data

Deletion with a cursor is done using the WHERE CURRENT OF clause in a DELETE statement. In general, the FOR UPDATE clause is not required for deletion of the current row of a cursor. The only exception occurs when using dynamic SQL (see Chapter 3, “Writing Dynamic Programs” on page 91 for information on dynamic SQL) for either the SELECT statement or the DELETE statement in an application which has been precompiled with LANGLEVEL set to SAA1, and bound with BLOCKING ALL. In this case, a FOR UPDATE clause is necessary in the SELECT statement. See the *Command Reference* for information on the precompiler options.

The DELETE statement causes the row being referenced by the cursor to be deleted. This leaves the cursor positioned before the *next* row and a FETCH statement must be issued before additional WHERE CURRENT OF operations may be performed against the cursor.

Types of Cursors

Cursors fall into three categories:

Read only The rows in the cursor can only be read, not updated. Read-only cursors are used when an application will only read data, not modify it. A cursor is considered read only if it is based on a read-only select-statement. See the rules in “Updating Retrieved Data” on page 63 for select-statements which define non-updatable result tables.

There can be performance advantages for read-only cursors. See “Row Blocking” on page 152 for more information.

Updatable The rows in the cursor can be updated. Updatable cursors are used when an application modifies data as the rows in the cursor are fetched. The specified query can only refer to one table or view. The query must also include the FOR UPDATE clause, naming each column that will be updated (unless the LANGLEVEL MIA precompile option is used).

Ambiguous The cursor cannot be determined to be updatable or read only from its definition or context. This can happen when a dynamic SQL statement is encountered that could be used to change a cursor that would otherwise be considered read-only.

An ambiguous cursor is treated as read only if the BLOCKING ALL option is specified when precompiling or binding. Otherwise, it is considered updatable.

Note: Cursors processed dynamically are always ambiguous.

For a complete list of criteria used to determine whether a cursor is read-only, updatable, or ambiguous, see the *SQL Reference*.

Example OPENFTCH Program

This example selects from a table using a cursor, opens the cursor, and fetches rows from the table. For each row fetched, it decides if the row should be deleted or updated (based on a simple criteria).

How the Example OPENFTCH SQL Program Works

1. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor c1 to a query. The query identifies the rows that the application retrieves using the FETCH statement. The job field of staff is defined to be updatable, even though it is not specified in the result table.
2. **Open the cursor.** The cursor c1 is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
3. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *current* row.
4. **Update OR Delete the current row.** The current row is either updated or deleted, depending upon the value of dept returned with the FETCH statement.

   If an UPDATE is performed, the position of the cursor remains on this row because the UPDATE statement does not change the position of the current row.

   If a DELETE statement is performed, a different situation arises, because the *current* row is deleted. This is equivalent to being positioned before the *next* row, and a FETCH statement must be issued before additional WHERE CURRENT OF operations are performed.
5. **Close the cursor.** The CLOSE statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl.

FORTRAN CHECKERR is a subroutine located in the util.f file.

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Example: OPENFTCH.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: OPENFTCH\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: openftch [userid passwd]\n\n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR   1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;   2
   CHECKERR ("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :pname, :dept;   3
      if (SQLCODE != 0) break;

      if (dept > 40) {
         printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
         EXEC SQL UPDATE staff SET job = 'Clerk'   4
            WHERE CURRENT OF c1;
         CHECKERR ("UPDATE STAFF");
      } else {
         printf ("%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
         EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
         CHECKERR ("DELETE");
      } /* endif */
   } while ( 1 );

   EXEC SQL CLOSE c1;   5
   CHECKERR ("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   CHECKERR ("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : OPENFTCH.SQC */
```

COBOL Example: OPENFTCH.SQB

```
 Identification Division.
 Program-ID. "openftch".

 Data Division.
 Working-Storage Section.
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 pname             pic x(10).
 01 dept              pic s9(4) comp-5.
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: OPENFTCH".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR                                1
              SELECT name, dept FROM staff
              WHERE job='Mgr'
              FOR UPDATE OF job END-EXEC.

     EXEC SQL OPEN c1 END-EXEC                                     2
     move "OPEN" to errloc.
     call "checkerr" using SQLCA errloc.

* call the FETCH and UPDATE/DELETE loop.
     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

     EXEC SQL CLOSE c1 END-EXEC.                                   5
```

```
    move "CLOSE" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL ROLLBACK END-EXEC.
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.
    display "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.  3
    if SQLCODE not equal 0
       go to End-Fetch-Loop.

    if dept greater than 40
       go to Update-Staff.

Delete-Staff.
    display pname, " in dept. ", dept,
       " will be DELETED!".

    EXEC SQL DELETE FROM staff WHERE CURRENT OF c1 END-EXEC.
    move "DELETE" to errloc.
    call "checkerr" using SQLCA errloc.

    go to End-Fetch-Loop.

Update-Staff.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".

    EXEC SQL UPDATE staff SET job = 'Clerk'                      4
             WHERE CURRENT OF c1 END-EXEC.
    move "UPDATE" to errloc.
    call "checkerr" using SQLCA errloc.

End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

FORTRAN Example: OPENFTCH.SQF

```
      program openftch
      implicit none

      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA
      EXEC SQL BEGIN DECLARE SECTION
        character*10  pname
        integer*2     dept
        character*8   userid
        character*18  passwd
      EXEC SQL END DECLARE SECTION

      character*80    errloc

      print *, 'Sample FORTRAN program: OPENFTCH'

      print *, 'Enter your user id (default none):'
      read 101, userid
101   format (a8)

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO sample
      else
      print *, 'Enter your password :'
      read 101, passwd

      EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL DECLARE c1 CURSOR FOR  1
     c         SELECT name, dept FROM staff WHERE job='Mgr'
     c         FOR UPDATE OF job

      EXEC SQL OPEN c1  2
      errloc = 'OPEN'
      call checkerr (sqlca, errloc, *999)

  100 continue
      EXEC SQL FETCH c1 INTO :pname, :dept  3
      if (sqlcode .ne. 0) goto 200

      if (dept .gt. 40) then
        print *, pname, ' in dept. ', dept, ' will be demoted to Clerk.'
        EXEC SQL UPDATE staff SET job = 'Clerk'  4
     c           WHERE CURRENT OF c1
        errloc = 'UPDATE'
        call checkerr (sqlca, errloc, *999)
      endif

      if (dept .lt. 41) then
        print *, pname, ' in dept. ', dept, ' will be DELETED!'
        EXEC SQL DELETE FROM staff WHERE CURRENT OF c1
        errloc = 'DELETE'
        call checkerr (sqlca, errloc, *999)
      endif
```

```
    goto 100

200 EXEC SQL CLOSE c1  5
    errloc = 'CLOSE'
    call checkerr (sqlca, errloc, *999)


    EXEC SQL ROLLBACK
    errloc = 'ROLLBACK'
    call checkerr (sqlca, errloc, *999)
    print *, 'On second thought -- changes rolled back.'

    EXEC SQL CONNECT RESET
    errloc = 'CONNECT RESET'
    call checkerr (sqlca, errloc, *999)

999 stop
    end
```

Advanced Scrolling Techniques

The following topics on advanced scrolling techniques are discussed in this section:

- Scroll through Data that has Already Been Retrieved
- Keeping a Copy of the Data
- Retrieving the Data a Second Time
- Establish a Position at the End of a Table
- Update Previously Retrieved Data

Scroll through Data that has Already Been Retrieved

When an application retrieves data from the database, the FETCH statement allows it to scroll forward through the data, however, the database manager has no embedded SQL statement that allows it scroll backwards through the data, (equivalent to a backward FETCH). DB2 CLI however, does support a backward FETCH through read-only scrollable cursors. See the *CLI Guide and Reference* for more information on scrollable cursors. For embedded SQL applications, you can use the following techniques to scroll through data that has been retrieved:

1. Keep a copy of the data that has been fetched and scroll through it by some programming technique.
2. Use SQL to retrieve the data again, typically by a second SELECT statement.

These options are discussed in more detail in:

- Keeping a Copy of the Data
- Retrieving the Data a Second Time

Keeping a Copy of the Data

An application can save fetched data in virtual storage. If the data does not fit in virtual storage, the application can write the data to a temporary file. One effect of this approach is that a user, scrolling backward, always sees exactly the same data that was fetched, even if the data in the database was changed in the interim by a transaction.

Using an isolation level of repeatable read, the data you retrieve from a transaction can be retrieved again by closing and opening a cursor. Other applications are prevented from updating the data in your result set. Isolation levels and locking can affect how users update data. For information on isolation levels, locks, and how they are related, see “Concurrency” on page 133.

Retrieving the Data a Second Time

This technique depends on the order in which you want to see the data again:

- Retrieving from the Beginning
- “Retrieving from the Middle” on page 73
- Order of Rows in the Second Result Table
- Retrieving in Reverse Order

Retrieving from the Beginning

To retrieve the data again from the beginning, merely close the active cursor and reopen it. This action positions the cursor at the beginning of the result table. But, unless the application holds locks on the table, others may have changed it, so what had been the first row of the result table may no longer be.

Retrieving from the Middle

To retrieve data a second time from somewhere in the middle of the result table, execute a second SELECT statement and declare a second cursor on the statement. For example, suppose the first SELECT statement was:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

Now, suppose that you want to return to the rows that start with `DEPTNO = 'M95'` and fetch sequentially from that point. Code the following:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'M95'
  ORDER BY DEPTNO
```

This statement positions the cursor where you want it.

Order of Rows in the Second Result Table

The rows of the second result table may not be displayed in the same order as in the first. The database manager does not consider the order of rows as significant unless the SELECT statement uses ORDER BY. Thus, if there are several rows with the same `DEPTNO` value, the second SELECT statement may retrieve them in a different order from the first. The only guarantee is that they will all be in order by department number, as demanded by the clause ORDER BY DEPTNO.

The difference in ordering could occur even if you were to execute the same SQL statement, with the same host variables, a second time. For example, the statistics in the catalog could be updated between executions, or indexes could be created or dropped. You could then execute the SELECT statement again.

The ordering is more likely to change if the second SELECT has a predicate that the first did not have; the database manager could choose to use an index on the new predicate. For example, it could choose an index on `LOCATION` for the first statement in our example and an index on `DEPTNO` for the second. Because rows are fetched in order by the index key, the second order need not be the same as the first.

Again, executing two similar SELECT statements can produce a different ordering of rows, even if no statistics change and no indexes are created or dropped. In the example, if there are many different values of `LOCATION`, the database manager could choose an index on `LOCATION` for both statements. Yet changing the value of `DEPTNO` in the second statement to the following, could cause the database manager to choose an index on `DEPTNO`:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

Because of the subtle relationships between the form of an SQL statement and the values in this statement, never assume that two different SQL statements will return rows in the same order unless the order is uniquely determined by an ORDER BY clause.

Retrieving in Reverse Order

Ascending ordering of rows is the default. If there is only one row for each value of DEPTNO, then the following statement specifies a unique ascending ordering of rows:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

To retrieve the same rows in reverse order, specify that the order is descending, as in the following statement:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO DESC
```

A cursor on the second statement retrieves rows in exactly the opposite order from a cursor on the first statement. Order of retrieval is guaranteed only if the first statement specifies a unique ordering sequence.

For retrieving rows in reverse order, it can be useful to have two indexes on the DEPTNO column, one in ascending order and the other in descending order.

Establish a Position at the End of a Table

The database manager does not guarantee an order to data stored in a table; therefore, the end of a table is not defined. However, order is defined on the result of an SQL statement:

```
SELECT * FROM DEPARTMENT
  ORDER BY DEPTNO DESC
```

For this example, the following statement positions the cursor at the row with the highest DEPTNO value:

```
SELECT * FROM DEPARTMENT
  WHERE DEPTNO =
  (SELECT MAX(DEPTNO) FROM DEPARTMENT)
```

Note, however, that if several rows have the same value, the cursor is positioned on the first of them.

Update Previously Retrieved Data

To scroll backward and update data that was retrieved previously, you can use a combination of the techniques discussed in “Scroll through Data that has Already Been Retrieved” on page 72 and “Updating Retrieved Data” on page 63. You can do one of two things:

1. If you have a second cursor on the data to be updated and if the SELECT statement uses none of the restricted elements, you can use a cursor-controlled UPDATE statement. Name the second cursor in the WHERE CURRENT OF clause.
2. In other cases, use UPDATE with a WHERE clause that names all the values in the row or specifies the primary key of the table. You can execute one statement many times with different values of the variables.

Example UPDAT Program

The UPDAT program uses dynamic SQL to access the STAFF table in the SAMPLE database and changes all managers to clerks. Then the program reverses the changes by rolling back the unit of work.

How the Example UPDAT Program Works

1. **Define an SQLCA structure.** The INCLUDE SQLCA statement defines and declares an SQLCA structure, and defines `SQLCODE` as an element within the structure. The SQLCODE field of the SQLCA structure is updated with error information by the database manager after execution of SQL statements and database manager API calls.

   REXX applications have one occurrence of an SQLCA structure, named SQLCA, predefined for application use. It can be referenced without application definition.

2. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are used to pass data to and from the database manager. They are prefixed with a colon (:) when referenced in an SQL statement.

   REXX applications do not need to declare host variables, except in the case of LOB file reference variables and locators. Other than LOB file reference variables and locators, host variable data types and sizes are determined at run time when the variables are referenced.

3. **Connect to database.** The program connects to the `sample` database, and requests shared access to it. (It is assumed that a START DATABASE MANAGER API call or `db2start` command has been issued.) Other programs that connect to the same database using shared access are also granted access.

4. **Execute the UPDATE SQL statement.** The SQL statement is executed *statically* with the use of a host variable. The `job` column of the `staff` tables is set to the value of the host variable, where the `job` column has a value of `Mgr`.

5. **Execute the DELETE SQL statement** The SQL statement is executed *statically* with the use of a host variable. All rows that have a `job` column value equal to that of the specified host variable, (`jobUpdate/job-update/job_update`) are deleted.

6. **Execute the INSERT SQL statement** A row is inserted into the STAFF table. This insertion implements the use of a host variable which was set prior to the execution of this SQL statement.

7. **End the transaction.** End the *unit of work* with a ROLLBACK statement. The result of the SQL statement executed previously can be either made permanent using the COMMIT statement, or undone using the ROLLBACK statement. All SQL statements within the *unit of work* are affected.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl.

FORTRAN CHECKERR is a subroutine located in the util.f file.

REXX CHECKERR is a procedure located at bottom of the current program.

See "Using GET ERROR MESSAGE in Example Programs" on page 85 for the source code for this error checking utility.

C Example: UPDAT.SQC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlenv.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;  1

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;  2
      char statement[256];
      char userid[9];
      char passwd[19];
      char jobUpdate[6];
   EXEC SQL END DECLARE SECTION;

   printf( "\nSample C program:  UPDAT \n");

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
         CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd; 3
         CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: updat [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (jobUpdate, "Clerk");
   EXEC SQL UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr';  4
   CHECKERR ("UPDATE STAFF");
   printf ("All 'Mgr' have been demoted to 'Clerk'!\n" );

   strcpy (jobUpdate, "Sales");
   EXEC SQL DELETE FROM staff WHERE job = :jobUpdate;  5
   CHECKERR ("DELETE FROM STAFF");
   printf ("All 'Sales' people have been deleted!\n");

   EXEC SQL INSERT INTO staff
      VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0);   6
   CHECKERR ("INSERT INTO STAFF");
   printf ("New data has been inserted\n");

   EXEC SQL ROLLBACK;  7
   CHECKERR ("ROLLBACK");
   printf( "On second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : UPDAT.SQC */
```

COBOL Example: UPDAT.SQB

```
 Identification Division.
 Program-ID. "updat".

 Data Division.
 Working-Storage Section.
     copy "sql.cbl".
     copy "sqlenv.cbl".
     copy "sqlca.cbl".  1

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  2
 01 statement         pic x(80).
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 job-update        pic x(5).
     EXEC SQL END DECLARE SECTION END-EXEC.

* Local variables
 77 errloc          pic x(80).
 77 error-rc        pic s9(9) comp-5.
 77 state-rc        pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
* Use application specific bound instead of BUFFER-SZ
 77 buffer-size     pic s9(4) comp-5 value 1024.
 77 line-width      pic s9(4) comp-5 value 80.
 77 error-buffer    pic x(1024).
 77 state-buffer    pic x(1024).

 Procedure Division.
 Main Section.
     display "Sample COBOL program:  UPDAT".

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd  3
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "Clerk" to job-update.
     EXEC SQL UPDATE staff SET job=:job-update  4
              WHERE job='Mgr' END-EXEC.
```

```
    move "UPDATE STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "All 'Mgr' have been demoted to 'Clerk'!".

    move "Sales" to job-update.
    EXEC SQL DELETE FROM staff WHERE job=:job-update END-EXEC.  5
    move "DELETE FROM STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "All 'Sales' people have been deleted!".

    EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99,        6
             :job-update, 0, 0, 0) END-EXEC.
    move "INSERT INTO STAFF" to errloc.
    call "checkerr" using SQLCA errloc.

    display "New data has been inserted".

    EXEC SQL ROLLBACK END-EXEC.                                   7
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.

    DISPLAY "On second thought -- changes rolled back."

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.


End-Prog.
    stop run.
```

FORTRAN UNIX Example: UPDAT.SQF

```
      program updat
      implicit none

      include 'sql.f'
      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA  1

      EXEC SQL BEGIN DECLARE SECTION  2
        character*80   statement
        character*8    userid
        character*18   passwd
        character*5    job_update
      EXEC SQL END DECLARE SECTION

      character*80     errloc

      print *, 'Sample FORTRAN program: UPDAT'

      print *, 'Enter your user id (default none):'
      read 100, userid
100   format (a8)

      if( userid(1:1) .eq. ' ' ) then
       EXEC SQL CONNECT TO sample
      else
       print *, 'Enter your password :'
       read 100, passwd

       EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if  3
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      job_update='Clerk'
      EXEC SQL UPDATE staff SET job = :job_update
     +         WHERE job = 'Mgr'  4
      errloc = 'UPDATE STAFF'
      call checkerr (sqlca, errloc, *999)
      print *, 'All ''Mgr'' have been demoted to ''Clerk!'''

      job_update='Sales'
      EXEC SQL DELETE FROM STAFF WHERE job = :job_update  5
      errloc = 'DELETE FROM STAFF'
      call checkerr (sqlca, errloc, *999)
      print *, 'All ''Sales'' people have been deleted!'

      EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99,  6
     +         :job_update, 0, 0, 0)
      errloc = 'INSERT INTO STAFF'
      call checkerr (sqlca, errloc, *999)
      print *, 'New data has been inserted'

      EXEC SQL ROLLBACK  7
      errloc = 'ROLLBACK'
      call checkerr (sqlca, errloc, *999)
      print *, 'On second thought -- changes rolled back.'

      EXEC SQL CONNECT RESET
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

  999 stop
      end
```

REXX Example: UPDAT.CMD

Note: REXX programs cannot contain static SQL. This program is written with dynamic SQL.

```
/* REXX program UPDAT.CMD */

/* REXX programs can not use STATIC SQL, thus all SQL statements must be */
/* converted to DYNAMIC SQL                                              */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
if passwd = '' then do
  say 'USAGE : updat userid passwd'
  exit
end

say 'Sample REXX program: UPDAT.CMD'

call SQLEXEC 'CONNECT TO sample USER' userid 'USING' passwd  3
call CHECKERR 'CONNECT TO'

jobupdate = "'Clerk'"
st = "UPDATE staff SET job =" jobupdate "WHERE job = 'Mgr'"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  4
call CHECKERR 'UPDATE'
say "All 'Mgr' have been demoted to 'Clerk'!"

jobupdate = "'Sales'"
st = "DELETE FROM staff WHERE job =" jobupdate
call SQLEXEC 'EXECUTE IMMEDIATE :st'  5
call CHECKERR 'DELETE'
say "All 'Sales' people have been deleted!"

st = "INSERT INTO staff VALUES (999, 'Testing', 99," jobupdate ", 0, 0, 0)"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  6
call CHECKERR 'INSERT'
say 'New data has been inserted'

call SQLEXEC 'ROLLBACK'  7
call CHECKERR 'ROLLBACK'
say 'On second thought...changes rolled back.'
```

```
call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Diagnostic Handling and the SQLCA Structure

Applications issuing SQL statements and calling database manager APIs must properly check for error conditions by examining return codes and the SQLCA structure.

Return Codes

Most database manager APIs pass back a zero return code when successful. In general, a non-zero return code indicates that the secondary error handling mechanism, the SQLCA structure, may be corrupt. In this case, the called API is not executed. A possible cause for a corrupt SQLCA structure is passing an invalid address for the structure.

SQLCODE and SQLSTATE

Error information is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure, which is updated after every executable SQL statement and most database manager API calls.

A source file containing executable SQL statements can provide at least one SQLCA structure with the name `sqlca`. The SQLCA structure is defined in the SQLCA include file. Source files without embedded SQL statements, but calling database manager APIs, can also provide one or more SQLCA structures, but their names are arbitrary.

If your application is compliant with the FIPS 127-2 standard, you can declare the SQLSTATE and SQLCODE as host variables instead of using the SQLCA structure. For information on how to do this, refer to “SQLSTATE and SQLCODE Variables” on page 450 for C or C++ applications, “SQLSTATE and SQLCODE Variables” on page 472 for COBOL applications, or “SQLSTATE and SQLCODE Variables” on page 487 for FORTRAN applications.

An SQLCODE value of 0 means successful execution (with possible SQLWARN warning conditions). A positive value means that the statement was successfully executed but with a warning, as with truncation of a host variable. A negative value means that an error condition occurred.

An additional field, SQLSTATE, contains a standardized error code consistent across other IBM database products and across SQL92 conformant database managers. Practically speaking, you should use SQLSTATEs when you are concerned about portability since SQLSTATEs are common across many database managers.

The SQLWARN field contains an array of warning indicators, even if SQLCODE is zero. The first element of the SQLWARN array, SQLWARN0, contains a blank if all other elements are blank. SQLWARN0 contains a `W` if at least one other element contains a warning character.

See the *API Reference* for more information about the SQLCA structure, and the *Message Reference* for a listing of SQLCODE and SQLSTATE error conditions.

Note: If you want to develop applications that access various IBM RDBMS servers you should:

- Where possible, have your applications check the SQLSTATE rather than the SQLCODE.
- If your applications will use DB2 Connect, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.

Handling Errors using the WHENEVER Statement

The WHENEVER statement causes the precompiler to generate source code that directs the application to go to a specified label if an error, warning, or if no rows are found during execution. The WHENEVER statement affects all subsequent executable SQL statements until another WHENEVER statement alters the situation.

The WHENEVER statement has three basic forms:

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

In the above statements:

SQLERROR Identifies any condition where SQLCODE < 0.

SQLWARNING Identifies any condition where SQLWARN(0) = W or SQLCODE > 0 but is not equal to 100.

NOT FOUND Identifies any condition where SQLCODE = 100.

In each case, the *action* can be either of the following:

CONTINUE Indicates to continue with the next instruction in the application.

GO TO *label* Indicates to go to the statement immediately following the label specified after GO TO. (GO TO can be two words, or one word, GOTO.)

If the WHENEVER statement is not used, the default action is to continue processing if an error, warning, or exception condition occurs during execution.

The WHENEVER statement must appear before the SQL statements you want to affect. Otherwise, the precompiler does not know that additional error-handling code should be generated for the executable SQL statements. You can have any combination of the three basic forms active at any time. The order in which you declare the three forms is not significant. To avoid an infinite looping situation, ensure that you undo the WHENEVER handling before any SQL statements are executed inside the handler. You can do this using the WHENEVER CONTINUE statement.

For a complete description of the WHENEVER statement, see the *SQL Reference*.

Exception, Signal, Interrupt Handler Considerations

An exception, signal, or interrupt handler is a routine that gets control when an exception, signal, or interrupt occurs. The type of handler applicable is determined by your operating environment, as shown in the following:

DOS Pressing `Ctrl-C` or `Ctrl-Break` generates an interrupt.

OS/2 Pressing `Ctrl-C` or `Ctrl-Break` generates an operating system exception.

UNIX Usually, pressing `Ctrl-C` generates the SIGINT interrupt signal. Note that keyboards can easily be redefined so SIGINT may be generated by a different key sequence on your machine.

For other operating systems that are not in the above list, consult the *DB2 SDK Building Applications* book for those operating systems.

Do not put SQL statements (other than COMMIT or ROLLBACK) in exception, signal, and interrupt handlers. With these kinds of error conditions, you normally want to do a ROLLBACK to avoid the risk of inconsistent data.

Note that you should exercise caution when coding a COMMIT and ROLLBACK in exception/signal/interrupt handlers. If you call either of these statements by themselves, the COMMIT or ROLLBACK is not executed until the current SQL statement is complete, if one is running. This is not the behavior desired from a `Ctrl-C` handler.

The solution is to call the INTERRUPT API (`sqleintr`/`sqlgintr`) before issuing a ROLLBACK. This interrupts the current SQL query (if the application is executing one) and lets the ROLLBACK begin immediately. If you are going to perform a COMMIT rather than a ROLLBACK, you do not want to interrupt the current command.

When using APPC to access a remote database server (DB2 for AIX or host database system using DB2 Connect), the application may receive a SIGUSR1 signal. This signal is generated by SNA Services/6000 when an unrecoverable error occurs and the SNA connection is stopped. You may want to install a signal handler in your application to handle SIGUSR1.

Refer to your platform documentation for specific details on the various handler considerations.

Exit List Routine Considerations

Do not use SQL or DB2 API calls in exit list routines. Note that you cannot disconnect from a database in an exit routine.

Using GET ERROR MESSAGE in Example Programs

The code clips shown in “C Example: UTIL.C” on page 86, “COBOL Example: CHECKERR.CBL” on page 88, and “FORTRAN Example: UTIL.F” on page 89, demonstrate the use of the GET ERROR MESSAGE API to obtain the corresponding information related to the SQLCA passed in.

You can find information on building these examples in the README files, or in the header section of these sample programs.

C Example: UTIL.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlenv.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>


#define  SQLSTATE sqlca.sqlstate

#ifndef max
#define  max(A, B) ((A) > (B) ? (A) : (B))
#endif
#ifndef min
#define  min(A, B) ((A) > (B) ? (B) : (A))
#endif

#define  CHECKERR(CE_STR)       check_error (CE_STR, &sqlca)
#define  LOBLENGTH  29

#if defined(DB2OS2)
   #include <conio.h>
#elif defined (DB2AIX)
   #include <curses.h>
#endif

struct lob_file {
   unsigned long name_length;
   unsigned long data_length;
   unsigned long file_option;
   char          name[255];
};

int check_error (char eString[], struct sqlca *caPointer) {
   char eBuffer[1024];
   char sBuffer[1024];
   short rc, Erc;

   if (caPointer->sqlcode != 0) {
      printf ("--- error report ---\n");
      printf ("ERROR occured : %s.\nSQLCODE : %ld\n", eString,
         caPointer->sqlcode);

      /**********************\
      * GET SQLSTATE MESSAGE *
      \**********************/
      rc = sqlogstt (sBuffer, 1024, 80, caPointer->sqlstate);

      /*****************************\
      * GET ERROR MESSAGE API called *
      \*****************************/
      Erc = sqlaintp (eBuffer, 1024, 80, caPointer);
      /* return code is the length of the eBuffer string */
      if (Erc > 0) printf ("%s", eBuffer);

      if (caPointer->sqlcode < 0) {
           if (rc == 0) {
            printf ("\n%s", sBuffer);
              }
         printf ("--- end error report ---\n");
         return 1;
      } else {
      /* errorCode is just a Warning message */
           if (rc == 0) {
            printf ("\n%s", sBuffer);
              }
         printf ("--- end error report ---\n");
         printf ("WARNING - CONTINUING PROGRAM WITH WARNINGS!\n");
         return 0;
      } /* endif */
   } /* endif */
   return 0;
}
  .
  .
  .
/* functions/procedures on setting up & outputting info from SQLDA */
  .
  .
  .
```

COBOL Example: CHECKERR.CBL

```
 Identification Division.
 Program-ID. "checkerr".

 Data Division.
 Working-Storage Section.
 copy "sql.cbl".

* Local variables
 77 error-rc        pic s9(9) comp-5.
 77 state-rc        pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
 77 buffer-size     pic s9(4) comp-5 value 1024.
 77 line-width      pic s9(4) comp-5 value 80.
 77 error-buffer    pic x(1024).
 77 state-buffer    pic x(1024).

 Linkage Section.
 copy "sqlca.cbl" replacing ==VALUE "SQLCA   "== by == ==
                            ==VALUE 136==        by == ==.
 01 errloc          pic x(80).

 Procedure Division using sqlca errloc.
 Checkerr Section.
     if SQLCODE equal 0
        go to End-Checkerr.

     display "--- error report ---".
     display "ERROR occured : ", errloc.
     display "SQLCODE : ", SQLCODE.

* GET ERROR MESSAGE API called *
     call "sqlgintp" using by value       buffer-size
                                          line-width
                           by reference   sqlca
                                          error-buffer
                     returning error-rc.

* GET SQLSTATE MESSAGE *
     call "sqlggstt" using by value       buffer-size
                                          line-width
                           by reference   sqlstate
                                          state-buffer
                     returning state-rc.
     if error-rc is greater than 0
        display error-buffer.


     if state-rc is greater than 0
        display state-buffer.

     if state-rc is less than 0
        display "return code from GET SQLSTATE =" state-rc.


     if SQLCODE is less than 0
        display "--- end error report ---"
        go to End-Prog.

     display "--- end error report ---"
     display "CONTINUING PROGRAM WITH WARNINGS!".
 End-Checkerr. exit program.
 End-Prog. stop run.
```

FORTRAN Example: UTIL.F

```
      subroutine checkerr (ca, errloc, *)
      character     ca(136)
      character*60  errloc

      include 'sqlenv.f'
      include 'sqlutil.f'
      include 'sqlca_cn.f'
      include 'sql.f'
      integer*4         error_rc
      integer*4         state_rc
      integer*2         line_width

      integer*2         buffer_size  /1024/
      character         error_buffer(1024)
      character         state_buffer(1024)

      error_rc = sqlgmcpy (%ref(sqlca),
     +                     %ref(ca),
     +                     %val(136))

      if ( sqlcode .ne. 0 ) then
        print *, '--- error report ---'
        print *, 'ERROR occured : ', errloc

*     GET ERROR MESSAGE *
        error_rc = sqlgintp ( %val(buffer_size),
     +                  %val(line_width),
     +                  %ref(sqlca),
     +                  %ref(error_buffer))

*     GET SQLSTATE MESSAGE *
        state_rc = sqlggstt (%val(buffer_size),
     +                       %val(line_width),
     +                       %ref(sqlstate),
     +                       %ref(state_buffer))
        if (state_rc .gt. 0) then
         print *, (state_buffer(i), i=1,state_rc)
        endif

        if (state_rc .lt. 0) then
          print *, 'return code from GET SQLSTATE = ', state_rc
        endif

        print *, 'SQLCODE : ', sqlcode
        if (error_rc .gt. 0) then
          print *, (error_buffer(i), i=1,error_rc)
          print *, '--- end error report ---'
          if (sqlcode .lt. 0) then
            return 1
          endif

          if (sqlcode .gt. 0) then
            print *, 'CONTINUING PROGRAM WITH WARNINGS'
            return
          endif
        endif

        print *, 'DID NOT GET ERROR MESSAGE'
        return 1
      endif
      return
      end
```

REXX Example: CHECKERR procedure within REXX Examples

```
/* ERROR checking procedure */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if rxfuncquery('SQLDBS') <> 0 then
    rcy = rxfuncadd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if rxfuncquery('SQLEXEC') <> 0 then
    rcy = rxfuncadd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")
  .
  .
  .
call CHECKERR 'PREPARE INSERT STATEMENT'
  .
  .
  .
CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Chapter 3. Writing Dynamic Programs

This chapter explains the capabilities and limitations of using the dynamic SQL to access your data. The following topics are discussed:

- Why Use Dynamic SQL?
- Using PREPARE, DESCRIBE, FETCH and the SQLDA
- Variable Input to Dynamic SQL
- The DB2 Call Level Interface (CLI)

Why Use Dynamic SQL?

You may want to use dynamic SQL when:

- You need all or part of the SQL statement to be generated during application execution.
- The objects referenced by the SQL statement do not exist at precompile time.
- You want the statement to always use the most optimal access path, based on current database statistics.
- You want to modify the compilation environment of the statement, that is, experiment with the special registers.

Dynamic SQL Support Statements

The dynamic SQL support statements accept a character-string host variable and a statement name as arguments. The host variable contains the SQL statement to be processed dynamically in text form. The statement text is not processed when an application is precompiled. In fact, the statement text does not have to exist at the time the application is precompiled. Instead, the SQL statement is treated as a host variable for precompilation purposes and the variable is referenced during application execution. These SQL statements are referred to as *dynamic* SQL.

Dynamic SQL support statements are required to transform the host variable containing SQL text into an executable form and operate on it by referencing the statement name. These statements are:

EXECUTE IMMEDIATE Prepares and executes a statement that does not use any host variables. All EXECUTE IMMEDIATE statements in an application are cached in the same place at run time, so only the last statement is known. Use this statement as an alternative to the PREPARE and EXECUTE statements.

PREPARE Turns the character string form of the SQL statement into an executable form of the statement, assigns a statement name, and optionally places information about the statement in an SQLDA structure.

EXECUTE Executes a previously prepared SQL statement. The statement can be executed repeatedly within a connection.

DESCRIBE Places information about a prepared statement into an SQLDA.

An application can execute most SQL statements dynamically. See Table 22 on page 523 for the complete list of supported SQL statements.

Note: The content of dynamic SQL statements follows the same syntax as static SQL statements, but with the following exceptions:

- Comments are not allowed.
- The statement cannot begin with EXEC SQL.
- The statement cannot end with the statement terminator. An exception to this is the CREATE TRIGGER statement which can contain a semicolon (;).

Comparing Dynamic SQL with Static SQL

The question of whether to use static or dynamic SQL for performance is usually of great interest to programmers. The answer, of course, is that it all depends on your situation. Refer to Table 2 to help you decide whether to use static or dynamic SQL. There may be certain considerations such as security which dictate static SQL, or your environment (such as whether you are using DB2 CLI or the CLP) which dictates dynamic SQL.

When making your decision, consider the following recommendations on whether to choose static or dynamic SQL in a particular situation. In the following table, 'either' means that there is no advantage to either static or dynamic SQL. Note that these are general recommendations only. Your specific application, its intended usage, and working environment dictate the actual choice. When in doubt, prototyping your statements as static SQL, then as dynamic SQL, and then comparing the differences is the best approach.

Table 2 (Page 1 of 2). Comparing Static and Dynamic SQL

Consideration	**Likely Best Choice**
Time to run the SQL statement: • Less than 2 seconds • 2 to 10 seconds • More than 10 seconds	 • Static • either • Dynamic
Data Uniformity • Uniform data distribution • Slight non-uniformity • Highly non-uniform distribution	 • Static • either • Dynamic
Range (<,>,BETWEEN,LIKE) Predicates • Very Infrequent • Occasional • Frequent	 • Static • either • Dynamic

Table 2 (Page 2 of 2). Comparing Static and Dynamic SQL

Consideration	**Likely Best Choice**
Repetitious Execution	
• Runs many times (10 or more times)	• either
• Runs a few times (less than 10 times)	• either
• Runs once	• Static
Nature of Query	
• Random	• Dynamic
• Permanent	• either
Run Time Environment (DML/DDL)	
• Transaction Processing (DML Only)	• either
• Mixed (DML and DDL - DDL affects packages)	• Dynamic
• Mixed (DML and DDL - DDL does not affect packages)	• either
(See paragraph following this table.)	
Frequency of RUNSTATS	
• Very infrequently	• Static
• Regularly	• either
• Frequently	• Dynamic

In general, an application using dynamic SQL has a higher start-up (or initial) cost per SQL statement due to the need to compile the SQL statements prior to using them. Once compiled, the execution time for dynamic SQL compared to static SQL should be equivalent and, in some cases, faster due to better access plans being chosen by the optimizer. Each time a dynamic statement is executed, the initial compilation cost becomes less of a factor. If multiple users are running the same dynamic application with the same statements, only the first application to issue the statement realizes the cost of statement compilation.

In a mixed DML and DDL environment, the compilation cost for a dynamic SQL statement may vary as the statement may be implicitly recompiled by the system while the application is running. In a mixed environment, the choice between static and dynamic SQL must also factor in the frequency in which packages are invalidated. If the DDL does invalidate packages, dynamic SQL may be more efficient as only those queries executed are recompiled when they are next used. Others are not recompiled. For static SQL, the entire package is rebound once it has been invalidated.

Now suppose, your particular application contains a mixture of the above characteristics and some of these characteristics suggest that you use static while others suggest dynamic. In this case, there is no clear cut decision and you should probably use whichever method you have the most experience with, and with which you feel most comfortable. Note that the considerations in the above table are listed roughly in order of importance.

Note: Static and dynamic SQL each come in two types that make a difference to the DB2 optimizer. These are:

1. Static SQL containing no host variables

   This is an unlikely situation which you may see only for:

   - *Initialization* code
   - Novice training examples

   This is actually the best combination from a performance perspective in that there is no run-time performance overhead and yet the DB2 optimizer's capabilities can be fully realized.

2. Static SQL containing host variables

   This is the traditional *legacy* style of DB2 applications. It avoids the run time overhead of a PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be harnessed since it does not know the entire SQL statement. A particular problem exists with highly non-uniform data distributions.

3. Dynamic SQL containing no parameter markers

   This is the typical style for random query interfaces (such as the CLP) and is the optimizer's preferred flavor of SQL. For complex queries, the overhead of the PREPARE statement is usually worthwhile due to improved execution time. For more information on parameter markers, see "Using Parameter Markers" on page 119.

4. Dynamic SQL containing parameter markers

   This is the most common type of SQL for CLI applications. The key benefit is that the presence of parameter markers allows the cost of the PREPARE to be amortized over the repeated executions of the statement, typically a select or insert. This amortization is true for all repetitive dynamic SQL applications. Unfortunately, just like static SQL with host variables, parts of the DB2 optimizer will not work since complete information is unavailable. The recommendation is to use *static SQL with host variables* or *dynamic SQL without parameter markers* as the most efficient options.

Using PREPARE, DESCRIBE, FETCH and the SQLDA

With static SQL, host variables used in embedded SQL statements are known at application compile time. With dynamic SQL, the embedded SQL statements and consequently the host variables are not known until application run time. Thus, for dynamic SQL applications, you need to deal with the list of host variables that are used in your application. You can use the DESCRIBE statement to obtain host variable information for any SELECT statement that has been prepared (using PREPARE), and store that information into the SQL descriptor area (SQLDA).

When the DESCRIBE statement gets executed in your application, the database manager defines your host variables in an SQLDA. Once the host variables are defined in the SQLDA, you can use the FETCH statement to assign values to the host variables, using a cursor.

For complete information on the PREPARE, DESCRIBE, and FETCH statements, and a description of the SQLDA, see the *SQL Reference*.

For an example of a simple dynamic SQL program that uses the PREPARE, DESCRIBE, and FETCH statements without using an SQLDA, see “Example Dynamic SQL Program” on page 96. For an example of a dynamic SQL program that uses the PREPARE, DESCRIBE, and FETCH statements and an SQLDA to process interactive SQL statements, see “Example ADHOC Program” on page 113.

Declaring and Using Cursors

Processing a cursor dynamically is nearly identical to processing it using static SQL. When a cursor is declared, it is associated with a query.

In the static SQL case, the query is a SELECT statement in text form, as shown in Figure 9 on page 56.

In the dynamic SQL case, the query is associated with a statement name assigned in a PREPARE statement. Any referenced host variables are represented by parameter markers. Figure 10 shows a DECLARE statement associated with a dynamic SELECT statement.

Language	Example Source Code

C/C++	`strcpy( prep_string, "SELECT tabname FROM syscat.tables"`
	`                      "WHERE tabschema = ?" );`
	`EXEC SQL PREPARE s1 FROM :prep_string;`
	`EXEC SQL DECLARE c1 CURSOR FOR s1;`
	EXEC SQL OPEN c1 USING :*host_var*;

COBOL	`MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`
	`     TO PREP-STRING.`
	`EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC.`
	`EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.`
	EXEC SQL OPEN C1 USING :*host-var* END-EXEC.

FORTRAN	`prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?'`
	`EXEC SQL PREPARE s1 FROM :prep_string`
	`EXEC SQL DECLARE c1 CURSOR FOR s1`
	EXEC SQL OPEN c1 USING :*host_var*

Figure 10. Declare Statement Associated with a Dynamic SELECT

The main difference between a static and a dynamic cursor is that a static cursor is prepared at precompile time, and a dynamic cursor is prepared at run time. Additionally, host variables referenced in the query are represented by parameter markers, which are replaced by run-time host variables when the cursor is opened.

For more information about how to use cursors, see the following sections:

- “Selecting Multiple Rows Using a Cursor” on page 55
- “Example Cursor Program” on page 58
- “Using Cursors in REXX” on page 500

Example Dynamic SQL Program

This sample program shows the processing of a cursor based upon a dynamic SQL statement. It lists all the tables in SYSCAT.TABLES except for the tables with the value STAFF in the name column.

How the Example Dynamic SQL Program Works

1. **Declare host variables.** This section includes declarations of three host variables:

table_name	Used to hold the data returned during the FETCH statement
st	Used to hold the dynamic SQL statement in text form
parm_var	Supplies a data value to replace the parameter marker in st.

2. **Prepare the statement.** An SQL statement with one parameter marker (indicated by '?') is copied to the host variable. This host variable is passed to the PREPARE statement for validation. The PREPARE statement parses the SQL text and prepares an access section for the package in the same way that the precompiler or binder does, only it happens at run time instead of during preprocessing.
3. **Declare the cursor.** The DECLARE statement associates a cursor with a dynamically prepared SQL statement. If the prepared SQL statement is a SELECT statement, a cursor is necessary to retrieve the rows from the result table.
4. **Open the cursor.** The OPEN statement initializes the cursor declared earlier to point before the first row of the result table. The USING clause specifies a host variable to replace the parameter marker in the prepared SQL statement. The data type and length of the host variable must be compatible with the associated column type and length.
5. **Retrieve the data.** The FETCH statement is used to move the NAME column from the result table into the table_name host variable. The host variable is printed before the program loops back to fetch another row.
6. **Close the cursor.** The CLOSE statement closes the cursor and releases the resources associated with it.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl.

FORTRAN CHECKERR is a subroutine located in the util.f file.

REXX CHECKERR is a procedure located at bottom of the current program.

See "Using GET ERROR MESSAGE in Example Programs" on page 85 for the source code for this error checking utility.

C Example: DYNAMIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char  table_name[19];
      char  st[80];   1
      char  parm_var[19];
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: DYNAMIC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
         CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: dynamic [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy( st, "SELECT tabname FROM syscat.tables" );
   strcat( st, " WHERE tabname <> ?" );
   EXEC SQL PREPARE s1 FROM :st;   2
   CHECKERR ("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;   3

   strcpy( parm_var, "STAFF" );
   EXEC SQL OPEN c1 USING :parm_var;   4
   CHECKERR ("OPEN");
   do {
      EXEC SQL FETCH c1 INTO :table_name;   5
      if (SQLCODE != 0) break;

      printf( "Table = %s\n", table_name );
   } while ( 1 );

   EXEC SQL CLOSE c1;   6
   CHECKERR ("CLOSE");

   EXEC SQL COMMIT;
   CHECKERR ("COMMIT");

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : DYNAMIC.SQC */
```

COBOL Example: DYNAMIC.SQB

```
 Identification Division.
 Program-ID. "dynamic".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 table-name      pic x(20).
 01 st              pic x(80).  1
 01 parm-var        pic x(18).
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: DYNAMIC".

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "SELECT TABNAME FROM SYSCAT.TABLES
-       " WHERE TABNAME <> ?" to st.
     EXEC SQL PREPARE s1 FROM :st END-EXEC.  2
     move "PREPARE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.  3

     move "STAFF" to parm-var.
     EXEC SQL OPEN c1 USING :parm-var END-EXEC.  4
     move "OPEN" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.  6
    move "CLOSE" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL COMMIT END-EXEC.
    move "COMMIT" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :table-name END-EXEC.  5
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display "TABLE = ", table-name.
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

FORTRAN UNIX Example: DYNAMIC.SQF

```
      program dynamic
      implicit none

      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION  1
        character*8   userid
        character*18  passwd
        character*20  table_name
        character*80  st
        character*18  parm_var
      EXEC SQL END DECLARE SECTION

        character*80  errloc

      print *, 'Sample FORTRAN program: DYNAMIC'

      print *, 'Enter your user id (default none):'
      read 101, userid
101   format (a8)

      if( userid(1:1) .eq. ' ' ) then
        EXEC SQL CONNECT TO sample
      else
        print *, 'Enter your password :'
        read 101, passwd

        EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      st = 'SELECT tabname FROM syscat.tables WHERE tabname <> ?'
      EXEC SQL PREPARE s1 FROM :st  2
      errloc = 'PREPARE'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL DECLARE c1 CURSOR FOR s1  3

      parm_var = 'STAFF'
      EXEC SQL OPEN c1 USING :parm_var  4
      errloc = 'OPEN'
      call checkerr (sqlca, errloc, *999)

   10 continue
         EXEC SQL FETCH c1 INTO :table_name  5
         if (sqlcode .ne. 0) goto 100
         print *, 'Table = ', table_name
      goto 10

  100 EXEC SQL CLOSE c1  6
      errloc = 'CLOSE'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL COMMIT
      errloc = 'COMMIT'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL CONNECT RESET
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

  999 stop
      end
```

REXX Example: DYNAMIC.CMD

```
/* REXX DYNAMIC.CMD */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
if passwd = '' then do
  say 'USAGE : dynamic userid passwd'
  exit
end

say 'Sample REXX program: DYNAMIC'

call SQLEXEC 'CONNECT TO sample USER' userid 'USING' passwd
call CHECKERR 'CONNECT TO'

st = "SELECT tabname FROM syscat.tables WHERE tabname <> ?"
call SQLEXEC 'PREPARE s1 FROM :st'   2
call CHECKERR 'PREPARE'

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'   3
call CHECKERR 'DECLARE'

parm_var = "STAFF"
call SQLEXEC 'OPEN c1 USING :parm_var'   4

do while ( SQLCA.SQLCODE = 0 )
  call SQLEXEC 'FETCH c1 INTO :table_name'   5
  if (SQLCA.SQLCODE = 0) then
    say 'Table = ' table_name
end

call SQLEXEC 'CLOSE c1'   6
call CHECKERR 'CLOSE'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Declaring the SQLDA

An SQLDA contains a variable number of occurrences of SQLVAR entries, each of which contains a set of fields that describe one column in a row of data as shown in Figure 11. There are two types of SQLVAR entries: base SQLVARs, and secondary SQLVARs. For information about the two types, see the *SQL Reference*.

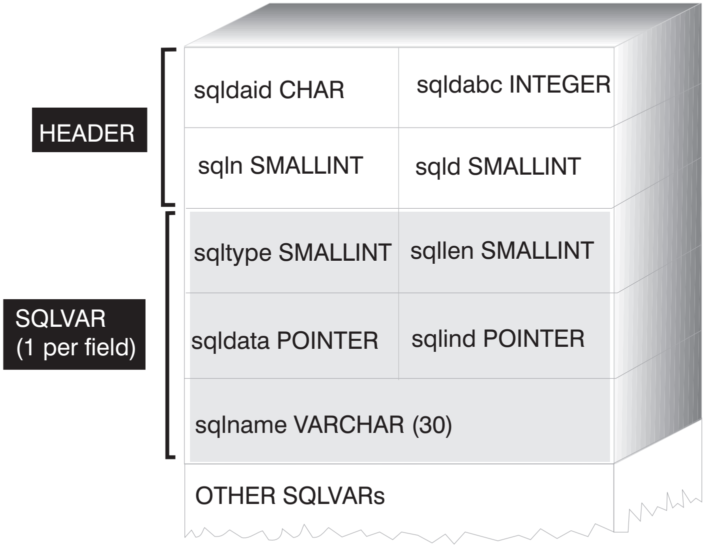


Figure 11. The SQL Descriptor Area (SQLDA)

Since the number of SQLVAR entries required depends on the number of columns in the result table, an application must be able to allocate an appropriate number of SQLVAR elements when needed. Two methods are available as discussed below. For information about the fields of the SQLDA that are mentioned, see the *SQL Reference*.

- Provide the largest SQLDA (that is, the one with the greatest number of SQLVAR entries) that is needed. The maximum number of columns that can be returned in a result table is 255. If any of the columns being returned is either a LOB type or a distinct type, the value in SQLN is doubled, and the number of SQLVARs needed to hold the information is doubled to 510. However, as most SELECT statements do not even retrieve 255 columns, most of the allocated space is unused.
- Provide a smaller SQLDA with fewer SQLVAR entries. In this case, if there are more columns in the result than SQLVAR entries allowed for in the SQLDA, then no descriptions are returned. Instead, the database manager returns the number of select list items detected in the SELECT statement. The application allocates an SQLDA with the required number of SQLVAR entries, and then uses the DESCRIBE statement to acquire the column descriptions. More details on this method are provided in “Preparing the Statement Using the Minimum SQLDA Structure” on page 104.

For the above methods, the question arises as to how many initial SQLVAR entries you should allocate. Each SQLVAR element uses up 44 bytes of storage (not counting storage allocated for the SQLDATA and SQLIND fields). If memory is plentiful, the first method of providing an SQLDA of maximum size is easier to implement.

The second method of allocating a smaller SQLDA is only applicable to programming languages such as C and C++ that support the dynamic allocation of memory. For languages such as COBOL and FORTRAN that do not support the dynamic allocation of memory, you have to use the first method.

Preparing the Statement Using the Minimum SQLDA Structure

Suppose an application declares an SQLDA structure named `minsqlda` that contains no SQLVAR entries. The SQLN field of the SQLDA describes the number of SQLVAR entries that are allocated. In this case, SQLN must be set to 0. Next, to prepare a statement from the character string `dstring` and to enter its description into `minsqlda`, issue the following SQL statement (assuming C syntax, and assuming that `minsqlda` is declared as a pointer to an SQLDA structure):

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

Suppose that the statement contained in `dstring` was a SELECT statement that returns 20 columns in each row. After the PREPARE statement (or a DESCRIBE statement), the SQLD field of the SQLDA contains the number of columns of the result table for the prepared SELECT statement.

The SQLVARs in the SQLDA are set in the following cases:

- SQLN >= SQLD and no column is either a LOB or a distinct type

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.

- SQLN >= 2*SQLD and at least one column is a LOB or a distinct type

  2* SQLD SQLVAR entries are set and SQLDOUBLED is set to `2`.

- SQLD <= SQLN < 2*SQLD and at least one column is a distinct type but there are no LOB columns

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no column is either a LOB or distinct type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVARs for a successful DESCRIBE.

- SQLN < SQLD and at least one column is a distinct type but there are no LOB columns

No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct types.

- SQLN < 2*SQLD and at least one column is a LOB

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

The SQLWARN option of the BIND command is used to control whether the DESCRIBE (or PREPARE...INTO) will return the following warnings:

- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005).

It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 (SQLSTATE 01005) is always returned when there are LOB columns in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB column in the result set.

Allocating an SQLDA with Sufficient SQLVAR Entries

After the number of columns in the result table is determined, storage can be allocated for a second, full-size SQLDA. For example, if the result table contains 20 columns (none of which are LOB columns), a second SQLDA structure, `fulsqlda`, must be allocated with at least 20 SQLVAR elements (or 40 elements if the result table contains any LOBs or distinct types). For the rest of this example, assume that no LOBs or distinct types are in the result table.

The storage requirements for SQLDA structures consist of the following:

- A fixed-length header, 16 bytes in length, containing fields such as SQLN and SQLD
- A varying-length array of SQLVAR entries, of which each element is 44 bytes in length.

The number of SQLVAR entries needed for `fulsqlda` was specified in the SQLD field of `minsqlda`. This value was 20. Therefore, the storage allocation required for `fulsqlda` used in this example is:

```
16 + (20 * 44)
```

This value represents the size of the header plus 20 times the size of each SQLVAR entry, giving a total of 896 bytes.

You can use the SQLDASIZE macro to avoid doing your own calculations and to avoid any version-specific dependencies.

Describing the SELECT Statement

Having allocated sufficient space for fulsqlda, an application must take the following steps:

1. Store the value 20 in the SQLN field of fulsqlda.
2. Obtain information about the SELECT statement using the second SQLDA structure, fulsqlda. Two methods are available:
   - Use another PREPARE statement specifying fulsqlda instead of minsqlda.
   - Use the DESCRIBE statement specifying fulsqlda.

Using the DESCRIBE statement is preferred because the costs of preparing the statement a second time are avoided. The DESCRIBE statement simply reuses information previously obtained during the prepare operation to fill in the new SQLDA structure. The following statement can be issued:

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

After this statement is executed, each SQLVAR element contains a description of one column of the result table.

Acquiring Storage to Hold a Row

Before fetching any rows of the result table using an SQLDA structure, an application must do the following:

1. Analyze each SQLVAR description to determine how much space is required for the value of that column.

   Note that for Large Object (LOB) values, when the SELECT is described, the data type given in the SQLVAR is SQL_TYP_*x*LOB. This data type corresponds to a plain LOB host variable, that is, the whole LOB will be stored in memory at one time. This will work for small LOBs (up to a few MB), but you cannot use this data type for large LOBs (say 1 GB). It will be necessary for your application to change its column definition in the SQLVAR to be either SQL_TYP_*x*LOB_LOCATOR or SQL_TYPE_*x*LOB_FILE. (Note that changing the SQLTYPE field of the SQLVAR also necessitates changing the SQLLEN field.) After changing the column definition in the SQLVAR, your application can then allocate the correct amount of storage for the new type. For more information on LOBs, see Chapter 6, “Using the Object-Relational Capabilities” on page 229.
2. Allocate storage for the value of that column.
3. Store the address of the allocated storage in the SQLDATA field of the SQLDA structure.

These steps are accomplished by analyzing the description of each column and replacing the content of each SQLDATA field with the address of a storage area large enough to hold any values from that column. The length attribute is determined from the SQLLEN field of each SQLVAR entry for data items that are not of a LOB type. For items with a type of BLOB, CLOB, or DBCLOB, the length attribute is determined from the SQLLONGLEN field of the secondary SQLVAR entry.

In addition, if the specified column allows nulls, then the application must replace the content of the SQLIND field with the address of an indicator variable for the column.

Processing the Cursor

After the SQLDA structure is properly allocated, the cursor associated with the SELECT statement can be opened and rows can be fetched by specifying the USING DESCRIPTOR clause of the FETCH statement.

When finished, the cursor should be closed and any dynamically allocated memory should be released.

Allocating an SQLDA Structure

To create an SQLDA structure with C, either embed the INCLUDE SQLDA statement in the host language or include the SQLDA include file to get the structure definition. Then, because the size of an SQLDA is not fixed, the application must declare a pointer to an SQLDA structure and allocate storage for it. The actual size of the SQLDA structure depends on the number of distinct data items being passed using the SQLDA. (For an example of how to code an application to process the SQLDA, see “Example ADHOC Program” on page 113.)

In the C/C++ programming language, a macro is provided to facilitate SQLDA allocation. This macro has the following format:

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1) * sizeof(struct sqlvar))
```

The effect of this macro is to calculate the required storage for an SQLDA with `n` SQLVAR elements.

To create an SQLDA structure with COBOL, you can either embed an INCLUDE SQLDA statement or use the COPY statement. Use the COPY statement when you want to control the maximum number of SQLVARs and hence the amount of storage that the SQLDA uses. For example, to change the default number of SQLVARs from 1489 to 1, use the following COPY statement:

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

The FORTRAN language does not directly support self-defining data structures or dynamic allocation. No SQLDA include file is provided for FORTRAN, because it is not possible to support the SQLDA as a data structure in FORTRAN. The precompiler will ignore the INCLUDE SQLDA statement in a FORTRAN program.

However, you can create something similar to a static SQLDA structure in a FORTRAN program, and use this structure wherever an SQLDA can be used. The file `sqldact.f` contains constants that help in declaring an SQLDA structure in FORTRAN.

Execute calls to SQLGADDR to assign pointer values to the SQLDA elements that require them.

The following table shows the declaration and use of an SQLDA structure with one SQLVAR element.

Language — **Example Source Code**

C/C++

```
#include <sqlda.h>
struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));

/* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */
double sal;
short salind;

/* INITIALIZE ONE ELEMENT OF SQLDA */
memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));
outda->sqln = outda->sqld = 1;
outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;
outda->sqlvar[0].sqllen  = sizeof( double );.
outda->sqlvar[0].sqldata = (unsigned char *)&sal;
outda->sqlvar[0].sqlind  = (short *)&salind;
```

COBOL

```
 WORKING-STORAGE SECTION.
 77 SALARY          PIC S99999V99 COMP-3.
 77 SAL-IND         PIC S9(4)     COMP-5.

    EXEC SQL INCLUDE SQLDA END-EXEC

* Or code a useful way to save unused SQLVAR entries.
* COPY "sqlda.cbl" REPLACING --1489-- BY --1--.

      01 decimal-sqllen pic s9(4) comp-5.
      01 decimal-parts redefines decimal-sqllen.
         05 precision pic x.
         05 scale pic x.

* Initialize one element of output SQLDA
      MOVE 1 TO SQLN
      MOVE 1 TO SQLD
      MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)

* Length = 7 digits precision and 2 digits scale



      MOVE x"07" TO PRECISION.
      MOVE x"02" TO SCALE.
      MOVE DECIMAL-SQLLEN TO O-SQLLEN(1).



      SET SQLDATA(1) TO ADDRESS OF SALARY
      SET SQLIND(1)  TO ADDRESS OF SAL-IND
```

Language **Example Source Code**

FORTRAN

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     Declare an Output SQLDA -- 1 Variable
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! Header
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! First Variable
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                              )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                              )

C     Declare Local Variables for Holding Returned Data.
      real*8       salary
      integer*2    sal_ind

C     Initialize the Output SQLDA (Header)
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     Initialize VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

In languages not supporting dynamic memory allocation, an SQLDA with the desired number of SQLVAR elements must be explicitly declared in the host language. Be sure to declare enough SQLVAR elements as determined by the needs of the application.

Passing Data Using an SQLDA Structure

Greater flexibility is available when passing data using an SQLDA than is available using lists of host variables. For example, an SQLDA can be used to transfer data that has no native host language equivalent, such as DECIMAL data in the C language. The sample program called ADHOC is an example using this technique. (See "Example ADHOC Program" on page 113.) See Table 3 for a convenient cross-reference listing showing how the numeric values and symbolic names are related.

Table 3 (Page 1 of 2). DB2 V2 SQLDA SQL Types. Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name[1]
DATE	384/385	SQL_TYP_DATE / SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME / SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP / SQL_TYP_NSTAMP
n/a[2]	400/401	SQL_TYP_CGSTR / SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB / SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB / SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR / SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG / SQL_TYP_NLONG
n/a[3]	460/461	SQL_TYP_CSTR / SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGRAPH / SQL_TYP_NLONGRAPH
FLOAT	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
REAL[4]	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL[5]	484/485	SQL_TYP_DECIMAL / SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER / SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL / SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE
n/a	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR

Table 3 (Page 2 of 2). DB2 V2 SQLDA SQL Types. Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name[1]

Note: These defined types can be found in the `sql.h` include file located in the `include` sub-directory of the `sqllib` directory. (For example, `sqllib/include/sql.h` for the C programming language.)

1. For the COBOL programming language, the SQLTYPE name does not use underscore (_) but uses a hyphen (-) instead.
2. This is a null-terminated graphic string.
3. This is a null-terminated character string.
4. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. Precision is in the first byte. Scale is in the second byte.

Processing Interactive SQL Statements

An application using dynamic SQL can be written to process arbitrary SQL statements. For example, if an application accepts SQL statements from a user, the application must be able to execute the statements without any prior knowledge of the statements.

By using the PREPARE and DESCRIBE statements with an SQLDA structure, an application can determine the type of SQL statement being executed, and act accordingly.

For an example of a program that processes interactive SQL statements, see “Example ADHOC Program” on page 113.

Determining Statement Type

When an SQL statement is prepared, information concerning the type of statement can be determined by examining the SQLDA structure. This information is placed in the SQLDA structure either at statement preparation time with the INTO clause, or by issuing a DESCRIBE statement against a previously prepared statement.

In either case, the database manager places a value in the SQLD field of the SQLDA structure, indicating the number of columns in the result table generated by the SQL statement. If the SQLD field contains a zero (0), the statement is *not* a SELECT statement. Since the statement is already prepared, it can immediately be executed using the EXECUTE statement.

If the statement contains parameter markers, the USING clause must be specified as described in *SQL Reference*. The USING clause can specify either a list of host variables or an SQLDA structure.

If the SQLD field is greater than zero, the statement is a SELECT statement and must be processed as described in the following sections.

Varying-List SELECT Statement

A *varying-list* SELECT statement is one in which the number and types of columns that are to be returned are not known at precompilation time. In this case, the application does not know in advance the exact host variables that need to be declared to hold a row of the result table.

To process a varying-list SELECT statement, an application can do the following:

1. **Declare an SQLDA.** An SQLDA structure must be used to process varying-list SELECT statements.
2. **PREPARE the statement using the INTO clause.** The application then determines whether the SQLDA structure declared has enough SQLVAR elements. If it does not, the application allocates another SQLDA structure with the required number of SQLVAR elements, and issues an additional DESCRIBE statement using the new SQLDA.
3. **Allocate the SQLVAR elements.** Allocate storage for the host variables and indicators needed for each SQLVAR. This step involves placing the allocated addresses for the data and indicator variables in each SQLVAR element.
4. **Process the SELECT statement.** A cursor is associated with the prepared statement, opened, and rows are fetched using the properly allocated SQLDA structure.

These steps are described in detail in the following sections:

- "Declaring the SQLDA" on page 103
- "Preparing the Statement Using the Minimum SQLDA Structure" on page 104
- "Allocating an SQLDA with Sufficient SQLVAR Entries" on page 105
- "Describing the SELECT Statement" on page 106
- "Acquiring Storage to Hold a Row" on page 106
- "Processing the Cursor" on page 107.

Saving SQL Requests from End Users

If your application allows users to save arbitrary SQL statements, you can save them in a table with a column having a data type of VARCHAR, LONG VARCHAR, CLOB, VARGRAPHIC, LONG VARGRAPHIC or DBCLOB. Note that the VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data types are only available in Double Byte Character Support (DBCS) and Extended UNIX Code (EUC) environments.

You must save the source SQL statements, not the prepared versions. This means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your application prepares an SQL statement from a character string and executes this statement dynamically.

Example ADHOC Program

This sample program shows how the SQLDA is used to process interactive SQL statements.

Note: This example exists for C only.

How the Example ADHOC Program Works

1. **Define an SQLDA structure.** The INCLUDE SQLDA statement defines and declares an SQLDA structure, which is used to pass data from the database manager to the program and back.
2. **Define an SQLCA structure.** The INCLUDE SQLCA statement defines an SQLCA structure, and defines SQLCODE as an element within the structure. The SQLCODE field of the SQLCA structure is updated with diagnostic information by the database manager after execution of SQL statements.
3. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement.
4. **Connect to database.** The program connects to the database specified by the user, and requests shared access to it. (It is assumed that a START DATABASE MANAGER API call or db2start command has been issued.) Other programs that attempt to connect to the same database in share mode are also granted access.
5. **Check completion.** The SQLCA structure is checked for successful completion of the CONNECT TO statement. An SQLCODE value of 0 indicates that the connection was successful.
6. **Interactive prompt.** SQL statements are entered in through the prompt and then are sent to the process_statement function for further processing.
7. **End the transaction - COMMIT.** The unit of work is ended with a COMMIT if so chosen by the user. All changes requested by the SQL statements entered since this last COMMIT are saved in the database.
8. **End the transaction - ROLLBACK.** The unit of work is ended with a ROLLBACK if so chosen by the user. All changes requested by the SQL statements entered since the last COMMIT or the start of the program, are undone.
9. **Disconnect from the database.** The program disconnects from the database by executing the CONNECT RESET statement. Upon return, the SQLCA is checked for successful completion.
10. **Copy SQL statement text to host variable.** The statement text is copied into the data area specified by the host variable st.
11. **Prepare the SQLDA for processing.** An initial SQLDA structure is declared and memory is allocated through the init_da procedure to determine what type of output the SQL statement could generate. The SQLDA returned from this PREPARE statement reports the number of columns that will be returned from the SQL statement.

12. **SQLDA reports output columns exist.** The SQL statement is a SELECT statement. The SQLDA is initialized through the init_da procedure to allocate memory space for the prepared SQL statement to reside in.

13. **SQLDA reports no output columns.** There are no columns to be returned. The SQL statement is executed *dynamically* using the EXECUTE statement.

14. **Preparing memory space for the SQLDA.** Memory is allocated to reflect the column structures in the SQLDA. The required amount of memory is selected by the SQLTYPE and the SQLLEN of the column structure in the SQLDA.

15. **Declare and open a cursor.** The DECLARE statement associates the cursor pcurs with the dynamically prepared SQL statement in sqlStatement and the cursor is opened.

16. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the SQLDA.

17. **Display the column titles.** The first row that is fetched is the column title information.

18. **Display the row information.** The rows of information collected from each consecutive FETCH is displayed.

19. **Close the cursor.** The CLOSE statement is closes the cursor, and releases the resources associated with it.

The CHECKERR macro/function is an error checking utility which is external to this program. In this program, check_error is redefined as CHECKERR and is located in the util.c file. See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

Note that this example uses a number of additional procedures that are provided as utilities in the file util.c. These include:

get_info Returns the input from a passed parameter but only if the input conforms to a specified sizing restriction, thus it is used to verify input retrieval.

init_da Allocates memory for a prepared SQL statement. An internally described function called SQLDASIZE is used to calculate the proper amount of memory.

alloc_host_vars Allocates memory for data from an SQLDA pointer.

free_da Frees up the memory that has been allocated to use an SQLDA data structure.

print_var Prints out the SQLDA SQLVAR variables. This procedure first determines data type then calls the appropriate subroutines that are required to print out the data.

display_da Displays the output of a pointer that has been passed through. All pertinent information on the structure of the output data is available from this pointer, as examined in the procedure print_var.

C Example: ADHOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>
#include <sqlda.h>  1
#include "util.h"

EXEC SQL INCLUDE SQLCA;  2

/* 'check_error' is a function found in the util.c program */
#define  CHECKERR(CE_STR)       check_error (CE_STR, &sqlca)
#define  SQLSTATE sqlca.sqlstate

int process_statement (char[1000]);

int main(void) {
   char  rc;
   char  sqlInput[255];
   char st[1000];
   char Transaction;

   EXEC SQL BEGIN DECLARE SECTION;  3
      char  server[9];
      char  userid[9];
      char  passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf("Sample C program : ADHOC interactive SQL\n");

   /* Initialize the connection to a database. */
   do {
      printf("input the server (database) which you wish to attach to : \n");
      gets (sqlInput);
      strcpy(server,sqlInput);

      printf("input your userid : \n");
      gets (sqlInput);
      strcpy (userid, sqlInput);

      printf("input your passwd : \n");
      gets (sqlInput);
      strcpy (passwd, sqlInput);

      printf("CONNECTING TO %s\n",server);
      EXEC SQL CONNECT TO :server USER :userid USING :passwd;  4
      CHECKERR("CONNECT TO DATABASE");  5
   } while (SQLCODE != 0); /* enddo */

   printf("CONNECTED TO %s\n",server);

   /* Enter the continuous command line loop. */
   while ( 1 ) {  6
      printf ("Enter an SQL statement or 'quit' to Quit :\n");
      gets (sqlInput);

      if (strcmp(sqlInput, "quit") == 0) {
         break;
      } else if (strlen(sqlInput) == 0) {
         /* Don't process the statement */
         printf ("\tNo characters entered.\n");
      } else if (sqlInput[strlen(sqlInput) - 1] == '\\') {
         /* Check to see if there are more lines that need to be appended
            to the command. */
         strcpy (st, "\0");
         do {
            strncat (st, sqlInput, strlen(sqlInput) -1);
            gets (sqlInput);
         } while (sqlInput[strlen(sqlInput) - 1] == '\\');
         strcat (st, sqlInput);

         /* Process the statement. */
         rc = process_statement (st);
      } else {
         strcpy (st, sqlInput);

         /* Process the statement. */
         rc = process_statement (st);
      } /* end if */
   } /* end while */

   printf ("Enter 'c' to COMMIT or Any Other key to "
      "ROLLBACK the transaction :\n");

   Transaction = getc(stdin);
   if (Transaction == 'c') {

      printf("COMMITING the transactions.\n");
      EXEC SQL COMMIT;  7
      CHECKERR ("COMMIT");
   } else {
      /* assume that the transaction is to be rolled back */
      printf("ROLLING BACK the transactions.\n");
      EXEC SQL ROLLBACK;  8
      CHECKERR ("ROLLBACK");
   }; /* endif */

   EXEC SQL CONNECT RESET;  9
   CHECKERR ("CONNECT RESET");
   return 0;
}

 /******************************************************************************
 * FUNCTION : process_statement
 * This function processes the inputted statement and then prepares the
 *  procedural SQL implementation to take place.
 ******************************************************************************/
int process_statement (char sqlInput[1000]) {
   int counter = 0;
   struct sqlda *sqldaPointer;
   EXEC SQL BEGIN DECLARE SECTION;  3
      char st[1000];
   EXEC SQL END DECLARE SECTION;

   strcpy(st, sqlInput);  10
   /* allocate an initial SQLDA temp pointer to obtain information about the
      inputted "st" */

   init_da (&sqldaPointer, 1);  11

   EXEC SQL PREPARE statement1 from :st;
   if (CHECKERR ("PREPARE") != 0) return SQLCODE;

   EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer;
   /* Expecting a return code of 0 or SQL_RC_W236, SQL_RC_W237, SQL_RC_W238,
      SQL_RC_W239 for cases where this statement is a SELECT statment. */
   if (SQLCODE != 0 &&
       SQLCODE != SQL_RC_W236 &&
       SQLCODE != SQL_RC_W237 &&
       SQLCODE != SQL_RC_W238 &&
       SQLCODE != SQL_RC_W239) {
      /* An unexpected warning/error has occured.  Check the SQLCA. */
      if (CHECKERR ("DESCRIBE") != 0) return SQLCODE;
   } /* end if */

   if (sqldaPointer->sqld > 0) {  12
      /* this is a SELECT statement, a number of columns are present in the
         SQLDA */
      /* need to string compare the SQLSTATE with "01005"
         If at least one column is a LOB and SQLN < 2*SQLD then SQLDOUBLED is
         set to the space character and a warning is issued (SQLSTATE 01005).
         No SQLVAR entries are set.
         To see if it is a doubled SQLDA or not, check the SQLSTATE */
      if (strncmp(SQLSTATE, "01005", sizeof(SQLSTATE)) == 0) {
         /* this output contains columns that need a DOUBLED SQLDA */
         SETSQLDOUBLED (sqldaPointer, SQLDOUBLED);
         init_da (&sqldaPointer, sqldaPointer->sqld * 2);
      } else {
         /* this out only needs a SINGLE SQLDA */
         init_da (&sqldaPointer, sqldaPointer->sqld);
      } /* end if */

      /* need to reassign the SQLDA with the correct number of columns to
         the SQL statement */
      EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer;
      if (CHECKERR ("DESCRIBE") != 0) return SQLCODE;

      /* allocating the proper amount of memory space needed for the
         variables */
      alloc_host_vars (sqldaPointer);  14

      /* Don't need to check the SQLCODE for declaration of cursors */
      EXEC SQL DECLARE pcurs CURSOR FOR statement1;  15

      EXEC SQL OPEN pcurs;  15
      if (CHECKERR ("OPEN") != 0) return SQLCODE;

      EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer;  16
      if (CHECKERR ("FETCH") != 0) return SQLCODE;

      /* if the FETCH is successful, obtain data from SQLDA */
      /* display the column titles */
      display_col_titles (sqldaPointer);  17

      /* display the rows that are fetched */
```

```
      while (SQLCODE == 0) {
         counter++;
         display_da (sqldaPointer);  18
         EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer;
      } /* endwhile */

      EXEC SQL CLOSE pcurs;  19
      if (CHECKERR ("CLOSE CURSOR") != 0) return SQLCODE;
      printf ("\n  %d record(s) selected\n\n", counter);

      /* Free the memory allocated to this SQLDA. */
      free_da(sqldaPointer);

   } else {   /* this is not a SELECT statement, execute SQL statement */  13
      EXEC SQL EXECUTE statement1;
      if (CHECKERR ("executing the SQL statement") != 0) return SQLCODE;
   } /* end if */

   return SQLCODE;
}
/* end of program : ADHOC.SQC */
```

Variable Input to Dynamic SQL

This section shows you how to use parameter markers in your dynamic SQL applications to represent host variable information. It includes:

- Using Parameter Markers
- Example VARINP Program

Using Parameter Markers

A dynamic SQL statement cannot contain host variables, because host variable information (data type and length) is available only during application precompilation. At execution time, the host variable information is not present. Therefore, a new method is needed to represent application variables. Host variables are represented by a question mark (?) which is called a *parameter marker*. Parameter markers indicate the places in which a host variable is to be substituted inside of an SQL statement. The parameter marker takes on an assumed data type and length that is dependent on the context of its use inside the SQL statement.

If the data type of a parameter marker is not obvious from the context of the statement in which it is used, the type can be specified using a CAST. Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like a host variable of the given type. For example, the statement `SELECT ? FROM SYSCAT.TABLES` is invalid because DB2 does not know the type of the result column. However, the statement `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES`, is valid because the cast *promises* that the parameter marker represents an INTEGER, so DB2 knows the type of the result column.

A character string containing a parameter marker might look like the following:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

When this statement is executed, a host variable or SQLDA structure is specified by the USING clause of the EXECUTE statement. The contents of the host variable are used when the statement executes.

If the SQL statement contains more than one parameter marker, then the USING clause of the EXECUTE statement must either specify a list of host variables (one for each parameter marker), or it must identify an SQLDA that has an SQLVAR entry for each parameter marker. (Note that for LOBs, there are two SQLVARs per parameter marker.) The host variable list or SQLVAR entries are matched according to the order of the parameter markers in the statement, and they must have compatible data types.

Note that using a parameter marker with dynamic SQL is like using host variables with static SQL. In either case, the optimizer does not use distribution statistics, and possibly may not choose the best access plan.

The rules that apply to parameter markers are listed under the PREPARE statement in the *SQL Reference*.

Example VARINP Program

This is an example of an UPDATE that uses a parameter marker in the search and update conditions.

How the Example VARINP SQL Program Works

1. **Prepare the SELECT SQL statement** The PREPARE statement is called to dynamically prepare an SQL statement. In this SQL statement, parameter markers are denoted by the ?. The job field of staff is defined to be updatable, even though it is not specified in the result table.
2. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor c1 to the query that was prepared in **1**.
3. **Open the cursor.** The cursor c1 is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
4. **Prepare the UPDATE SQL statement** The PREPARE statement is called to dynamically prepare an SQL statement. The parameter marker in this statement is set to be Clerk but can be changed dynamically to anything, as long as it conforms to the column data type it is being updated into.
5. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *CURRENT* row.
6. **Update the current row.** The current row and specified column, job, is updated with the content of the passed parameter parm_var.
7. **Close the cursor.** The CLOSE statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl

FORTRAN CHECKERR is a subroutine located in the util.f file.

REXX CHECKERR is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Example: VARINP.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
      char st[255];
      char parm_var[6];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: VARINP \n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: varinp [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (st, "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job");
   EXEC SQL PREPARE s1 FROM :st;  1
   CHECKERR ("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;  2

   strcpy (parm_var, "Mgr");
   EXEC SQL OPEN c1 USING :parm_var;  3
   CHECKERR ("OPEN");

   strcpy (parm_var, "Clerk");
   strcpy (st, "UPDATE staff SET job = ? WHERE CURRENT OF c1");
   EXEC SQL PREPARE s2 from :st;  4

   do {
      EXEC SQL FETCH c1 INTO :pname, :dept;  5
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
              pname, dept );
      EXEC SQL EXECUTE s2 USING :parm_var;  6
      CHECKERR ("EXECUTE");
   } while ( 1 );

   EXEC SQL CLOSE c1;   7
   CHECKERR ("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   CHECKERR ("ROLLBACK");
   printf( "\nOn second thought -- changes rolled back.\n" );

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : VARINP.SQC */
```

COBOL Example: VARINP.SQB

```
 Identification Division.
 Program-ID. "varinp".

 Data Division.
 Working-Storage Section.
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 pname             pic x(10).
 01 dept              pic s9(4) comp-5.
 01 st                pic x(127).
 01 parm-var          pic x(5).
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: VARINP".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "SELECT name, dept FROM staff
-         "   WHERE job = ? FOR UPDATE OF job" to st.
     EXEC SQL PREPARE s1 FROM :st END-EXEC.  1
     move "PREPARE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.  2

     move "Mgr" to parm-var.

     EXEC SQL OPEN c1 USING :parm-var END-EXEC  3
     move "OPEN" to errloc.
     call "checkerr" using SQLCA errloc.

     move "Clerk" to parm-var.
     move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.

     EXEC SQL PREPARE s2 from :st END-EXEC. 4
     move "PREPARE S2" to errloc.
     call "checkerr" using SQLCA errloc.

* call the FETCH and UPDATE loop.
     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

     EXEC SQL CLOSE c1 END-EXEC. 7
     move "CLOSE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL ROLLBACK END-EXEC.
     move "ROLLBACK" to errloc.
     call "checkerr" using SQLCA errloc.
     DISPLAY "On second thought -- changes rolled back.".

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.
 End-Main.
     go to End-Prog.

 Fetch-Loop Section.
     EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC. 5
     if SQLCODE not equal 0
        go to End-Fetch-Loop.
     display pname, " in dept. ", dept,
        " will be demoted to Clerk".

     EXEC SQL EXECUTE s2 USING :parm-var END-EXEC. 6
     move "EXECUTE" to errloc.
     call "checkerr" using SQLCA errloc.

 End-Fetch-Loop. exit.

 End-Prog.
     stop run.
```

FORTRAN Example: VARINP.SQF

```
      program varinp
      implicit none

      include 'sqlenv.f'
      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION  1
        character*8   userid
        character*18  passwd
        character*9   pname
        integer*4     dept
        character*127 st
        character*5   parm_var
      EXEC SQL END DECLARE SECTION

        character*80  errloc

      print *, 'Sample FORTRAN program: VARINP'

      print *, 'Enter your user id (default none):'
      read 109, userid
109   format (a8)

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO sample
      else
      print *, 'Enter your password :'
      read 109, passwd

      EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      st='SELECT name, dept FROM staff WHERE job=? FOR UPDATE of job'
      EXEC SQL PREPARE s1 FROM :st  1
      errloc = 'PREPARE'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL DECLARE c1 CURSOR FOR s1  2

      parm_var = 'Mgr'
      EXEC SQL OPEN c1 USING :parm_var  3
      errloc = 'OPEN'
      call checkerr (sqlca, errloc, *999)

      parm_var = 'Clerk'
      st = 'UPDATE staff SET job = ? WHERE CURRENT OF c1'
      EXEC SQL PREPARE s2 FROM :st  4

   10 continue
         EXEC SQL FETCH c1 INTO :pname, :dept  5
         if (sqlcode .ne. 0) goto 100

         print *, pname, 'in dept. ', dept, ' will be demoted to Clerk'

         EXEC SQL EXECUTE s2 USING :parm_var  6
```

```
    goto 10

100 EXEC SQL CLOSE c1  7
    errloc = 'CLOSE'
    call checkerr (sqlca, errloc, *999)

    EXEC SQL ROLLBACK
    errloc = 'ROLLBACK'
    call checkerr (sqlca, errloc, *999)
    print *, 'On second thought -- changes rolled back.'

    EXEC SQL CONNECT RESET
    errloc = 'CONNECT RESET'
    call checkerr (sqlca, errloc, *999)

999 stop
    end
```

The DB2 Call Level Interface (CLI)

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the database, and executed. In contrast, a DB2 CLI application does not have to be precompiled or bound, but instead uses a standard set of functions to execute SQL statements and related services at run time.

This difference is important because, traditionally, precompilers have been specific to each database product, which effectively ties your applications to that product. DB2 CLI enables you to write portable applications that are independent of any particular database product. This independence means DB2 CLI applications do not have to be recompiled or rebound to access different DB2 databases, including DRDA databases. They just connect to the appropriate database at run time.

Comparing Embedded SQL and DB2 CLI

DB2 CLI and embedded SQL also differ in the following ways:

- DB2 CLI does not require the explicit declaration of cursors. DB2 CLI has a supply of cursors that get used as needed. The application can then use the generated cursor in the normal cursor fetch model for multiple row `SELECT` statements and positioned `UPDATE` and `DELETE` statements.
- The `OPEN` statement is not used in DB2 CLI. Instead, the execution of a `SELECT` automatically causes a cursor to be opened.
- Unlike embedded SQL, DB2 CLI allows the use of parameter markers on the equivalent of the `EXECUTE IMMEDIATE` statement (the `SQLExecDirect()` function).
- A `COMMIT` or `ROLLBACK` in DB2 CLI is issued via the `SQLEndTran()` function call rather than by passing it as an SQL statement.
- DB2 CLI manages statement related information on behalf of the application, and provides a *statement handle* to refer to it as an abstract object. This handle eliminates the need for the application to use product specific data structures.
- Similar to the statement handle, the *environment handle* and *connection handle* provide a means to refer to all global variables and connection specific information. The *descriptor handle* describes either the parameters of an SQL statement or the columns of a result set.
- DB2 CLI uses the SQLSTATE values defined by the X/Open SQL CAE specification. Although the format and most of the values are consistent with values used by the IBM relational database products, there are differences. (There are also differences between ODBC SQLSTATES and the X/Open defined SQLSTATES).
- DB2 CLI supports read-only scrollable cursors. With scrollable cursors, you can scroll through a static read-only cursor as follows:
  - Forward by one or more rows
  - Backward by one or more rows
  - From the first row by one or more rows

- From the last row by one or more rows.

Despite these differences, there is an important common concept between embedded SQL and DB2 CLI: *DB2 CLI can execute any SQL statement that can be prepared dynamically in embedded SQL.*

Note: DB2 CLI can also accept some SQL statements that cannot be prepared dynamically, such as compound SQL statements.

Table 22 on page 523 lists each SQL statement, and indicates whether or not it can be executed using DB2 CLI. The table also indicates if the command line processor can be used to execute the statement interactively, (useful for prototyping SQL statements).

Each DBMS may have additional statements that you can dynamically prepare. In this case, DB2 CLI passes the statements to the DBMS. There is one exception: the COMMIT and ROLLBACK statement can be dynamically prepared by some DBMSs but are not passed. In this case, use the `SQLEndTran()` function to specify either the COMMIT or ROLLBACK statement.

Advantages of Using DB2 CLI

The DB2 CLI interface has several key advantages over embedded SQL.

- It is ideally suited for a client-server environment, in which the target database is not known when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application is connected to.
- It increases the portability of applications by removing the dependence on precompilers.
- Individual DB2 CLI applications do not need to be bound to each database, only bind files shipped with DB2 CLI need to be bound once for all DB2 CLI applications. This can significantly reduce the amount of management required for the application once it is in general use.
- DB2 CLI applications can connect to multiple databases, including multiple connections to the same database, all from the same application. Each connection has its own commit scope. This is much simpler using CLI than using embedded SQL where the application must make use of multi-threading to achieve the same result.
- DB2 CLI eliminates the need for application controlled, often complex data areas, such as the SQLDA and SQLCA, typically associated with embedded SQL applications. Instead, DB2 CLI allocates and controls the necessary data structures, and provides a *handle* for the application to reference them.
- DB2 CLI enables the development of multi-threaded thread-safe applications where each thread can have its own connection and a separate commit scope from the rest. DB2 CLI achieves this by eliminating the data areas described above, and associating all such data structures that are accessible to the application with a specific handle. Unlike embedded SQL, a multi-threaded CLI application does not

need to call any of the context management DB2 APIs; this is handled by the DB2 CLI driver automatically.

- DB2 CLI provides enhanced parameter input and fetching capability, allowing arrays of data to be specified on input, retrieving multiple rows of a result set directly into an array, and executing statements that generate multiple result sets.
- DB2 CLI provides a consistent interface to query catalog (Tables, Columns, Foreign Keys, Primary Keys, etc.) information contained in the various DBMS catalog tables. The result sets returned are consistent across DBMSs. This shields the application from catalog changes across releases of database servers, as well as catalog differences amongst different database servers; thereby saving applications from writing version specific and server specific catalog queries.
- Extended data conversion is also provided by DB2 CLI, requiring less application code when converting information between various SQL and C data types.
- DB2 CLI incorporates both the ODBC and X/Open CLI functions, both of which are accepted industry specifications. DB2 CLI is also aligned with the emerging ISO CLI standard. Knowledge that application developers invest in these specifications can be applied directly to DB2 CLI development, and vice versa. This interface is intuitive to grasp for those programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.
- DB2 CLI provides the ability to retrieve multiple rows and result sets generated from a stored procedure residing on a DB2 Universal Database (or DB2 for MVS/ESA version 5 or later) server. However, note that this capability exists for Version 5 DB2 Universal Database clients using embedded SQL if the stored procedure resides on server that is accessible from a DataJoiner Version 2 server.
- DB2 CLI supports server-side scrollable cursors that can be used in conjunction with array output. This is useful in GUI applications that display database information in scroll boxes that make use of the Page Up, Page Down, Home and End keys. You can declare a read-only cursor as scrollable then move forward or backward through the result set by one or more rows. You can also fetch rows by specifying an offset from:
  - The current row
  - The beginning or end of the result set
  - A specific row you have previously set with a bookmark.
- DB2 CLI applications can dynamically describe parameters in an SQL statement the same way that CLI and Embedded SQL applications describe result sets. This enables CLI applications to dynamically process SQL statements that contain parameter markers without knowing the data type of those parameter markers in advance. When the SQL statement is prepared, describe information is returned detailing the data types of the parameters.

Deciding on Embedded SQL or DB2 CLI

Which interface you choose depends on your application.

DB2 CLI is ideally suited for query-based graphical user interface (GUI) applications that require portability. The advantages listed above, may make using DB2 CLI seem like the obvious choice for any application. There is however, one factor that must be considered, the comparison between static and dynamic SQL. Only embedded applications can use static SQL.

Static SQL has several advantages:

- Performance

  Dynamic SQL is prepared at run time, static SQL is prepared at precompile time. As well as requiring more processing, the preparation step may incur additional network-traffic at run time. This additional step (and network-traffic), however, will not be required if the DB2 CLI application makes use of deferred prepare.

  It is important to note that static SQL will not always have better performance than dynamic SQL. Dynamic SQL can make use of changes to the database, such as new indexes, and can use current database statistics to choose the optimal access plan. In addition, precompilation of statements can be avoided if they are cached.
- Encapsulation and Security

  In static SQL, the authorizations to objects (such as a table, view) are associated with a package and are validated at package binding time. This means that database administrators need only to grant execute on a particular package to a set of users (thus encapsulating their privileges in the package) without having to grant them explicit access to each database object. In dynamic SQL, the authorizations are validated at run time on a per statement basis; therefore, users must be granted explicit access to each database object. This permits these users access to parts of the object that they do not have a need to access.
- Embedded SQL is supported in languages other than C or C++.
- For fixed query selects, embedded SQL is simpler.

If an application requires the advantages of both interfaces, it is possible to make use of static SQL within a DB2 CLI application by creating a stored procedure that contains the static SQL. The stored procedure is called from within a DB2 CLI application and is executed on the server. Once the stored procedure is created, any DB2 CLI or ODBC application can call it. For more information, refer to the *CLI Guide and Reference*.

It is also possible to write a mixed application that uses both DB2 CLI and embedded SQL, taking advantage of their respective benefits. In this case, DB2 CLI is used to provide the base application, with key modules written using static SQL for performance or security reasons. This complicates the application design, and should only be used if stored procedures do not meet the applications requirements. For more information, refer to the section on *Mixing Embedded SQL and DB2 CLI* in the *CLI Guide and Reference*.

Ultimately, the decision on when to use each interface, will be based on individual preferences and previous experience rather than on any one factor.

Chapter 4. Programming Considerations for Concurrency and Performance

This chapter discusses topics and programming techniques you should consider to develop efficient application programs. In addition, there are numerous database and database manager configuration parameters which can be adjusted to improve the performance of application programs, such as those controlling buffer pool sizes and space allocation for blocking. See the *Administration Guide* for information about how database configuration and database manager configuration affect application performance.

The following topics are discussed:

- Concurrency
- Locking
- Row Blocking
- Adjusting the Optimization Class
- Join Strategies in a Partitioned Database

Concurrency

The integrity of the data in a relational database must be maintained as multiple users access and change the data. *Concurrency* is the sharing of resources by multiple interactive users or application programs at the same time. The database manager controls this access to prevent undesirable effects, such as:

- *Lost updates*. Two applications, A and B, might both read the same row from the database and both calculate new values for one of its columns based on the data these applications read. If A updates the row with its new value and B then also updates the row, the update performed by A is lost.

- *Access to uncommitted data*. Application A might update a value in the database, and application B might read that value before it was committed. Then, if the value of A is not later committed, but backed out, the calculations performed by B are based on uncommitted (and presumably invalid) data.

- *Nonrepeatable reads*. Some applications involve the following sequence of events: application A reads a row from the database, then goes on to process other SQL requests. In the meantime, application B either modifies or deletes the row and commits the change. Later, if application A attempts to read the original row again, it receives the modified row or discovers that the original row has been deleted.

- *Phantom Read Phenomenon.* The phantom read phenomenon occurs when:

  1. Your application executes a query that reads a set of rows based on some search criterion.
  2. Another application inserts new data or updates existing data that would satisfy your application's query.
  3. Your application repeats the query from step 1 (within the same unit of work).

When the query is repeated (step 3), some additional ("phantom") rows are returned as part of the result set that were not returned when the query was initially executed (step 1).

An *isolation level* determines how data is locked or isolated from other processes while the data is being accessed. The isolation level will be in effect for the duration of the unit of work. Applications that use a cursor declared using the WITH HOLD clause will keep the chosen isolation level for the duration of the unit of work in which the OPEN CURSOR was performed. (For more information, refer to the *SQL Reference* manual.) See "Specifying the Isolation Level" on page 137 for information on how the isolation level is specified.

DB2 supports the following isolation levels:

- Repeatable Read
- Read Stability
- Cursor Stability
- Uncommitted Read.

(Note that some DRDA database servers support the *no commit* isolation level. On other databases, it behaves like the uncommitted read isolation level. Refer to the *SQL Reference* for information on this isolation level.)

See also:

- "Choosing the Isolation Level" on page 136
- "Specifying the Isolation Level" on page 137
- "Transactions" on page 138.

Repeatable Read

Repeatable read (RR) locks all the rows an application references within a unit of work. Using repeatable read, a SELECT statement issued by an application twice within the same unit of work, in which the cursor was opened, gives the same result each time. With repeatable read, lost updates, access to uncommitted data, and phantom rows are not possible.

The repeatable read application can retrieve and operate on the rows as many times as needed until the unit of work completes. However, no other applications can update, delete, or insert a row that would affect the result table, until the unit of work completes. Repeatable read applications cannot see uncommitted changes of other applications.

With repeatable read, every row that is referenced is locked, not just the rows that are retrieved. Appropriate locking is performed so that another application cannot insert or update a row that would be added to the list of rows referenced by your query, if the query was re-executed. This prevents phantom rows from occurring. This means that if you scan 10 000 rows and apply predicates to them, locks are held on all 10 000 rows, even though only 10 rows qualify.

Note: The repeatable read isolation level ensures that all returned data remains unchanged until the time the application *sees* the data, even when temporary tables or row blocking are used.

Since repeatable read may acquire and hold a considerable number of locks, these locks may exceed the number of locks available as a result of the *locklist* and *maxlocks* configuration parameters. (Refer to the *Administration Guide* for recommendations and information on setting these parameters.) In order to avoid lock escalation, the optimizer may elect to immediately acquire a single table level lock for an index scan, if it believes that lock escalation is very likely to occur. (See "Lock Escalation" on page 144 for a discussion of lock escalation.) This functions as though the database manager has issued a LOCK TABLE statement on your behalf. If you do not want a table level lock to be obtained ensure that enough locks are available to the transaction or use the Read Stability isolation level.

Read Stability

Read stability (RS) locks only those rows that an application retrieves within a unit of work. It ensures that any qualifying row read during a unit of work is not changed by other application processes until the unit of work completes, and that any row changed by another application process is not read until the change is committed by that process. That is, "nonrepeatable read" behavior is **not** possible.

Unlike repeatable read, with read stability, if your application issues the same query more than once, you may see additional *phantom* rows (the *phantom read phenomenon*). Recalling the example of scanning 10 000 rows, read stability only locks the rows that qualify. Thus, with read stability, only 10 rows are retrieved, and a lock is held only on those ten rows. Contrast this with repeatable read, where in this example, locks would be held on all 10 000 rows. The locks that are held can be share, next share, update, or exclusive locks. (For more information on lock attributes, see "Attributes of Locks" on page 140.)

Note: The read stability isolation level ensures that all returned data remains unchanged until the time the application *sees* the data, even when temporary tables or row blocking are used.

One of the objectives of the read stability isolation level is to provide both a high degree of concurrency as well as a stable view of the data. To assist in achieving this objective, the optimizer ensures that table level locks are not obtained until lock escalation occurs. (See "Lock Escalation" on page 144 for more information about lock escalation).

The read stability isolation level is best for applications that include all of the following:

- Operate in a concurrent environment
- Require qualifying rows to remain stable for the duration of the unit of work
- Do not issue the same query more than once within the unit of work, or do not require that the query get the same answer when issued more than once in the same unit of work.

Cursor Stability

Cursor stability (CS) locks any row accessed by a transaction of an application while the cursor is positioned on the row. This lock remains in effect until the next row is fetched or the transaction is terminated. However, if any data on a row is changed, the lock must be held until the change is committed to the database.

No other applications can update or delete a row that a cursor stability application has retrieved while any updatable cursor is positioned on the row. Cursor stability applications cannot see uncommitted changes of other applications.

Recalling the example of scanning 10 000 rows, if you use cursor stability, you will only have a lock on the row under your current cursor position. The lock is removed when you move off that row (unless you update that row).

With cursor stability, both nonrepeatable read and the phantom read phenomenon are possible. Cursor stability is the default isolation level and should be used when you want the maximum concurrency while seeing only committed rows from other applications.

Uncommitted Read

Uncommitted read (UR) allows an application to access uncommitted changes of other transactions. The application also does not lock other applications out of the row it is reading, unless the other application attempts to drop or alter the table. Uncommitted read works differently for read-only and updatable cursors.

Read-only cursors can access most uncommitted changes of other transactions. However, tables, views, and indexes that are being created or dropped by other transactions are not available while the transaction is processing. Any other changes by other transactions can be read before they are committed or rolled back.

Cursors that are updatable operating under the uncommitted read isolation level will behave as if the isolation level was cursor stability.

Recalling the example of scanning 10 000 rows, if you use uncommitted read, you do not acquire any row locks.

With uncommitted read, both nonrepeatable read behavior and the phantom read phenomenon are possible.

The uncommitted read isolation level is most commonly used for queries on read-only tables, or if you are only executing select-statements and you do not care whether you see uncommitted data from other applications.

Choosing the Isolation Level

Table 4 on page 137 summarizes the different isolation levels in terms of the undesirable effects described in “Concurrency” on page 133 .

Table 4. Summary of isolation levels

Isolation Level	**Access to Uncommitted Data**	**Nonrepeatable Reads**	**Phantom Read Phenomenon**
Repeatable Read (RR)	Not Possible	Not Possible	Not Possible
Read Stability (RS)	Not Possible	Not Possible	Possible
Cursor Stability (CS)	Not Possible	Possible	Possible
Uncommitted Read (UR)	Possible	Possible	Possible

Table 5 provides a simple heuristic that may help you choose an initial isolation level for your applications. Consider this table as a starting point, and refer to the previous discussions of the various levels for factors that might make another value more appropriate for your requirements.

Table 5. Guidelines for choosing an isolation level

Application Type	**High data stability required**	**High data stability not required**
Read-write transactions	RS	CS
Read-only transactions	RR	UR

Choosing the appropriate isolation level for an application is very important to avoid the phenomena that are intolerable for that application. The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application since the CPU and memory resources, required to obtain and free locks, vary with the isolation level. The potential for deadlock situations also varies with the isolation level.

Specifying the Isolation Level

The isolation level is specified at precompile time or when an application is bound to a database. For an application written in a supported compiled language, use the ISOLATION option of the command line processor PREP or BIND commands. The isolation level can also be specified by using the PREP or BIND APIs. If no isolation level is specified, the default of cursor stability is used.

If a bind file is created at precompile time, the isolation level is stored in the bind file. If no isolation level is specified at bind time, the default is the isolation level used during precompilation.

You can determine the isolation level of a package by executing the following query:

```
SELECT ISOLATION FROM SYSCAT.PACKAGES
  WHERE PKGNAME = 'XXXXXXXX'
  AND PKGSCHEMA = 'YYYYYYYY'
```

where *XXXXXXXX* is the name of the package and *YYYYYYYY* is the schema name of the package. Both of these names must be in all capital letters.

When a database is created, multiple bind files used to support the different isolation levels for SQL in REXX are bound to the database (on those servers that support REXX). See “Execution Requirements for REXX” on page 500 for more information about these bind files.

REXX and the command line processor connect to a database using a default isolation level of cursor stability. Changing to a different isolation level does not change the connection state. It must be executed in the CONNECTABLE AND UNCONNECTED state or in the IMPLICITLY CONNECTABLE state. (See the CONNECT TO statement in the *SQL Reference* for details about connection states.) You cannot be connected to a database when issuing this command.

The isolation level being used can be checked by a REXX application by checking the value of the SQLISL REXX variable. The value is updated every time the CHANGE SQLISL command is executed.

Transactions

A transaction is a sequence of SQL statements (possibly with intervening host language code) that the database manager treats as a whole.

Transaction Management

A transaction (also known as *unit of work*), is the basic building block the database manager uses to ensure that a database is in a consistent state. Any reading or writing of the database is done within a unit of work. A *point of consistency* (or commit point) is a point in time when all recoverable data an application accesses is consistent. It occurs when updates, inserts, and deletions are either committed to the physical database or rolled back (not committed and therefore discarded).

For example, a banking application might transfer funds from account A to account B. After the application subtracts the amount from account A, the two accounts are inconsistent; not until the amount is added to account B are they consistent again. When both steps are complete, the application can announce a point of consistency, and then make the changes available to other applications. This entire process constitutes one transaction.

Any application that successfully connects to a database automatically starts a unit of work. The application must end the unit of work by issuing either a COMMIT or a ROLLBACK statement, or by disconnecting from the database. Disconnecting will automatically do a COMMIT and close all cursors. For more information on cursors, see “Selecting Multiple Rows Using a Cursor” on page 55.

The COMMIT statement makes all changes made within the unit of work permanent, while the ROLLBACK statement removes all these changes from the database. Either of these statements usually frees locks held on data objects by the unit of work, and another unit of work can access that data. During the execution of an application, an explicit COMMIT statement can be issued to create a point of consistency, or an explicit ROLLBACK statement can be issued to restore data changed by SQL statements back to the state at its last commit point. Once a COMMIT or ROLLBACK has been issued, it cannot be stopped.

If the application ends abnormally while in the middle of a unit of work (this can occur while the application is not executing a database manager API or SQL statement), the unit of work is rolled back. Normally, an application is expected to end units of work on a timely basis so that other units of work can also access the same data. For information about coding transactions, see “Coding Transactions” on page 7. For information regarding transactions in which multiple databases are accessed, refer to “Distributed Unit of Work” on page 389.

The consistency of a database is also protected in the event of a system crash by the recovery process. The recovery process uses a log of transactions to ensure consistency.

Transaction Logging

Transaction logging is the process that records each change to a database to permit recovery. A transaction is a sequence of statements to the database manager that performs one complete change. One transaction constitutes a unit of work.

The database manager maintains a log of recent changes made to a database so that the recovery process can restore the database to a consistent state. The consistent state is defined as follows: at the time of the system error, all units of work that had successfully committed or rolled back are restored to that state, respectively; all units of work that were *inflight* (units of work that had made changes but not yet committed or rolled back) are rolled back.

The recovery log contains the entries that describe the changes to database objects. Each entry contains the information needed to roll back the transaction before the changed data is written to the database. Thus, although some work may be lost, the database is restored to a state where all changes to data have been made by a unit of work that ended with a COMMIT statement.

If you are logging large object (LOB) data, you have to consider the impact to performance. If you turn logging on for LOB data, your application’s performance will deteriorate and you may encounter problems related to the increased size of the log file. If you turn the logging off, your application’s performance improves; however, its recoverability is sacrificed. For more information on the merits of logging LOB data, refer to the *Administration Guide*.

Locking

The database manager provides concurrency control and prevents uncontrolled access by means of locks. A *lock* is a means of associating a database manager resource with an application to control how other applications can access the same resource. The application with which the resource is associated is said to hold or own the lock.

The database manager imposes locks to prohibit applications from accessing uncommitted data written by other applications (unless the uncommitted read isolation level is used). This principle protects data integrity (that is, the consistency and security of data). Locks can also prohibit the updating of rows (such as for a repeatable read application).

To satisfy data integrity, the database manager acquires locks implicitly, under database manager control. Except for the uncommitted read isolation level, it is never necessary for an application to request a lock explicitly to ensure that uncommitted data is hidden from other processes.

Because of the basic principle of locking, you do not need to take action to control locks in most cases. Still, applications acquire locks on the basis of certain general parameters. Knowledge of your local situation can help you make better use of your system resources by changing those parameters. To assist you, the following topics on locking are discussed:

- Attributes of Locks
- Locks and Application Performance
- Factors Affecting Locking
- LOCK TABLE Statement
- CLOSE CURSOR WITH RELEASE
- Summary of Locking Considerations

Attributes of Locks

Database manager locks have the following basic attributes:

Object The resource being locked. The only types of explicitly lockable objects are tables. The database manager also imposes locks on other types of resources, such as rows, tables and table spaces. The object being locked represents the *granularity* of the lock.

Duration The length of time a lock is held. Lock durations are affected by isolation levels which are discussed in “Concurrency” on page 133.

Mode The type of access allowed for the lock owner as well as the type of access permitted for concurrent users of the locked object. It is sometimes referred to as the *state* of the lock.

Modes and their effects are shown in order of increasing control over resources:

IN (Intent None)
The lock owner can read any data in the table, including uncommitted data, but cannot change any of it. No row locks are acquired by the lock owner. Other concurrent applications can read or update the table. Both table spaces and tables can be locked in this mode.

IS (Intent Share)
The lock owner can read data in the locked table, but not change this data. When an application holds the IS table lock, the application acquires an S or NS lock on each row read. In

either case, other applications can read or update the table. Both table spaces and tables can be locked in this mode.

NS (Next Key Share)
This lock is acquired on rows of a table, instead of a Share lock. The lock owner and all concurrent applications can read, but not change, the locked row. Only individual rows can be locked in NS mode. This lock is acquired in place of a share (S) lock on data that is read with the RS or CS isolation levels.

S (Share)
The lock owner and any concurrent applications can read, but not change, the locked data. Individual rows can be Share locked. If a table is Share locked, no row locks are acquired by the lock owner. Other concurrent applications can read the table. Both rows and tables can be locked in this mode.

IX (Intent Exclusive)
The lock owner and concurrent applications can read and change data in the table. When the owner reads data, it acquires an S, NS, X, or U lock on each row. It also acquires an X lock on each row that it updates. Other concurrent applications can both read and update the table. Both table spaces and tables can be locked in this mode.

SIX (Share with Intent Exclusive)
The lock owner can both read and change data in the table. The lock owner acquires X locks on the rows it updates, but does not acquire locks on rows that it reads. Other concurrent applications can read the table. Only a table object can be locked in this mode.

U (Update)
The lock owner can update data in the locked object and acquire X locks on the rows prior to updates. Other units of work can read the data, but cannot attempt to update it. Both rows and tables can be locked in this mode. When a table is locked in U mode, X row locks are obtained.

NX (Next Key Exclusive)
This lock is acquired on the next row when a row is deleted from an index or inserted into the index of a table. The lock owner can read but not change the locked row. Only individual rows can be locked in NX mode. This is similar to an X lock except that it is compatible with the NS lock.

NW (Next Key Weak Exclusive)
This lock is acquired on the next row when a row is inserted into the index of a non-catalog table. The lock owner can read but not change the locked row. Only individual rows can be locked in NW mode. This is similar to X and NX locks except that it is compatible with the W and NS locks.

X (Exclusive)

The lock owner can both read and change data in the locked object. Tables can be Exclusive locked, meaning that no row locks will be acquired. Only uncommitted read applications can access the locked table. Both rows and tables can be locked in this mode.

W (Weak Exclusive)

This lock is acquired on the row when a row is inserted into a non-catalog table. The lock owner can change the locked row. Only individual rows are locked in W mode. This lock is similar to an X lock except that it is compatible with the NW lock. Only uncommitted read applications can access the locked row.

Z (Superxclusive)

This lock is acquired on a table in certain conditions, such as when the table is altered or dropped, an index on the table is created or dropped, or a table is reorganized. No other concurrent application can read or update the table. Both table space and table objects can be locked in this mode.

Note that only tables and table spaces will obtain the “intent” lock modes. That is, intent locks are not obtained for rows.

Locks and Application Performance

Application programmers need to be aware of several related factors concerning the uses of locks and their effect on the performance of applications. These factors include the following:

- Concurrency and Granularity
- Lock Compatibility
- Lock Conversion
- Lock Escalation
- Lock Waits and Timeouts
- Deadlocks.

Concurrency and Granularity

A lock held by one application can prevent access by another application. Therefore, for maximum concurrency, a row level lock is better than a table lock. But locks require storage and processing time to manage. Therefore, for minimizing storage and processing time, one table lock is better than many row locks.

Lock Compatibility

Table 6 indicates whether a lock request is granted if another process holds or is requesting a lock on the same resource in a given state. A **no** indicates that the requestor must wait until all incompatible locks are released by other processes. Note that a timeout can occur when waiting for a lock. A **yes** indicates that the lock is granted (unless someone else is waiting for the resource).

Table 6. Lock Type Compatibility

	State of Held Resource												
State Being Requested	**none**	**IN**	**IS**	**NS**	**S**	**IX**	**SIX**	**U**	**NX**	**X**	**Z**	**NW**	**W**
none	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
IN	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
IS	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no	no	no
NS	yes	yes	yes	yes	yes	no	no	yes	yes	no	no	yes	no
S	yes	yes	yes	yes	yes	no	no	yes	no	no	no	no	no
IX	yes	yes	yes	no	no	yes	no	no	no	no	no	no	no
SIX	yes	yes	yes	no	no	no	no	no	no	no	no	no	no
U	yes	yes	yes	yes	yes	no	no	no	no	no	no	no	no
NX	yes	yes	no	yes	no	no	no	no	no	no	no	no	no
X	yes	yes	no	no	no	no	no	no	no	no	no	no	no
Z	yes	no	no	no	no	no	no	no	no	no	no	no	no
NW	yes	yes	no	yes	no	no	no	no	no	no	no	no	yes
W	yes	yes	no	no	no	no	no	no	no	no	no	yes	no

Abbreviations:

I	Intent
N	None
NS	Next Key Share
S	Share
NX	Next Key Exclusive
X	Exclusive
U	Update
Z	Super Exclusive
NW	Next Key Weak Exclusive
W	Weak Exclusive

For details of these lock types, refer to the discussion in "Attributes of Locks" on page 140.

Legend:

- yes - **grant** lock requested immediately
- no - **wait** for held lock to be released or timeout to occur

Assume that application A holds a lock on a table that application B also wants to access. The database manager requests, on behalf of application B, a lock of some particular mode. If the mode of the lock held by A permits the lock requested by B, the two locks (or modes) are said to be compatible.

If the lock mode requested for application B is not compatible with the lock held by application A, application B cannot continue. Instead, it must wait not only until application A releases its lock, but until *all* existing incompatible locks are released.

Lock Conversion

Lock conversion occurs when a process accesses a data object on which it already holds a lock, and the mode of access requires a more restrictive lock than the one already held. A process can hold only one lock on a data object at any time, although it can (indirectly through a query) request a lock many times on the same data object. The operation of changing the mode of the lock already held is called a *conversion*.

The conversion case for rows is simple: As an example, a conversion occurs if an X is needed and an S or U is held.

There are more distinct lock modes for tables than for rows. IX (Intent Exclusive) and S (Shared) locks are special cases, however. Neither S nor IX is considered to be more restrictive than the other, so if one of these is held and the other required, the resulting conversion is to a SIX (Share with Intent Exclusive) lock. All other conversions result in the requested lock mode becoming the mode of the lock held, if the requested mode is more restrictive.

A query to update a row can also produce a dual conversion. Suppose the row had been read through an index access and was locked as S. The table containing the row would have a covering intention lock. Suppose it is an IS rather than an IX. Then, if the row is subsequently changed, the table lock is converted to an IX, and the row to an X.

As a reminder, the application of locks usually takes place implicitly during the execution of a query. Understanding the kinds of locks obtained for different queries and table and index combinations can assist you in designing and tuning your application. See “Factors Affecting Locking” on page 147 for more information on this topic.

Lock Escalation

Lock escalation occurs when too many locks (of any type) are currently held.

Lock escalation can occur for a specific database agent if the agent exceeds its allocation of the lock list .

Such escalation is handled internally; the only externally detectable result might be a reduction in concurrent access on one or more tables. Normally, in a properly configured database, lock escalation occurs infrequently.

An example of lock escalation is when an application designer uses an index on a large table to increase performance and concurrency; however, the application accesses a large percentage of records in the table. The database manager is not able to predict (in this case) that so much of the table will be locked, and locks each record individually rather than only locking the table either S or X.

Sometimes, the process receiving the escalation request (internally) holds few or no record locks on any table. The reason for this escalation is that one process (or processes) can be holding many locks (although this amount is below the database configuration parameter of locks per process) but not quite enough to trigger the escalation request. The process might not request another lock or access the database again except to end the transaction. Then another process can request the lock or locks that trigger the escalation request.

If lock escalation reduces concurrency to an unacceptable level, you can do the following:

- Increase the number of locks allowed by increasing the value of the *maxlocks* and/or the *locklist* parameters in the database configuration file. Refer to the *Administration Guide* for recommendations and information on setting these parameters. This might be the choice if concurrent access to the table by other processes is most important. However, the overhead of obtaining record level locks can induce more delay to other processes than is saved by concurrent access to a table. (When changing these parameters in a partitioned database, ensure that the parameters are updated on all partitions).
- Locate and adjust the offending process (or processes), which may or may not be the one escalating or rolling back, and issue LOCK TABLE statements explicitly.
- Change the degree of isolation. Note that this may lead to decreased concurrency or reduced isolation.
- Increase the frequency of commits. This tends to reduce the number of locks in existence at a given time. For more information about isolation levels and concurrency, see “Concurrency” on page 133 .

Lock Waits and Timeouts

Without lock timeout detection, in an abnormal situation, your application may have to wait for a lock to be released. This might occur, for example, when a transaction is waiting for a lock held by another user's application, and the other user has left their workstation without performing some interaction to allow their application to commit their transaction which would release the lock. Obviously, this results in poorer application performance. To avoid *stalling* your program in such a case, you can use the *locktimeout* configuration parameter to set the maximum time that any application waits to obtain a lock.

Using this parameter helps avoid global deadlocks, especially in distributed unit of work (DUOW) applications. If the lock times out, that is, if the time that the lock request is pending is greater than the *locktimeout* value, your application receives an error and your transaction is rolled back. For example, if *program1* tries to acquire a lock which is already held by *program2*, *program1* returns SQLCODE -911 (SQLSTATE 40001) with reason code 68 if the timeout is expired. Refer to the *Administration Guide* for recommendations and information on setting the *locktimeout* parameter. The default value for *locktimeout* when a database is created is -1 which indicates lock timeout detection is turned off.

Deadlocks

In the database manager, contention for locks by processes using the database can result in deadlocks. For example, Process 1 locks table A in X (exclusive) mode and Process 2 locks table B in X mode; if Process 1 then tries to lock table B in X mode and Process 2 tries to lock table A in X mode, the processes will be in a deadlock. In a deadlock, both processes are suspended until their second lock request is granted, and neither request is granted until one of the processes performs a commit or rollback. This state remains indefinitely until an outside agent activates one of the processes and forces it to perform a rollback.

Deadlocks in the lock system are handled in the database manager by an asynchronous system background process called the deadlock detector. The deadlock detector becomes active periodically as determined by the *dlchktime* configuration parameter. When the deadlock detector becomes active, it examines the lock system for deadlocks. If the database has been partitioned then each partition sends *lock graphs* to the catalog node where global deadlock detection takes place.

If a deadlock is found, the deadlock detector selects a deadlocked process to roll back. The selected process is awakened, and it returns to the calling application with SQLCODE -911 (SQLSTATE 40001), with reason code 2. The database manager rolls back the selected process automatically. When the rollback has completed, the locks belonging to the victim process are released, and the other processes involved in the deadlock can eventually proceed.

Selecting the proper interval for the deadlock detector is necessary to ensure good performance. An interval that is too short would cause unnecessary overhead, and one that is too long would allow a deadlock to delay a process for an unacceptable amount of time. For example, a wakeup interval set to 30 minutes could allow a deadlock to exist for nearly 30 minutes. The application designer must balance the possible delays in resolving deadlocks with the overhead of detecting them. For information on setting the wakeup interval, refer to the *dlchktime* configuration parameter in the *Administration Guide*.

In a partitioned database, the interval should be the same on all partitions (the *dlchktime* configuration parameter must be updated to the same value on all partitions). If the value is smaller at the catalog node than at other partitions, phantom deadlocks may be detected. If the value is larger at the catalog node than at other partitions, it may appear as if more than two intervals pass before a deadlock is detected. If a large number of deadlocks are detected in a partitioned database, you should increase the value of the *dlchktime* parameter to account for lock waits and communication waits.

Another problem can occur when an application with more than one independent process accessing the database is structured in such a way as to make deadlocks likely. An example is an application in which several processes access the same table for reads and then writes. If the processes do read-only SQL queries at first and then do SQL updates on the same table, the chances of deadlocks occurring increase because of potential contention between the processes for the same data. For instance, if two processes read the table, and then update the table, they get into a state where process A is trying to get an X lock on a row, on which process B has an S lock and vice versa. The result could be a deadlock. To avoid these deadlocks, applications that access data with the intention of modifying it should use the FOR UPDATE OF clause when performing a select. This clause ensures that a U lock is imposed when process A attempts to read the data.

Factors Affecting Locking

The mode and granularity of database manager locks are determined by a combination of factors: the type of processing the application performs, how it accesses data, and several parameters that you can specify.

Application Processing

For the purpose of determining lock attributes, processing can be classified as one of four types:

Read-only This type includes all select-statements which are intrinsically read-only (refer to the *SQL Reference* for information about cursors), have an explicit FOR READ ONLY clause, or are ambiguous but for which the SQL compiler presumes to be read-only due to the value of the BLOCKING option specified on the PREP or BIND command. It requires only Share locks (S or IS).

Intent to change This type includes all select-statements with the FOR UPDATE clause, or which the SQL compiler presumes to be intended for change as a result of the interpretation of the ambiguous statement. It uses Share and Update locks (S, U, and X for rows, IX, U, X for tables).

Change This type includes UPDATE, INSERT, and DELETE, but not UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF. It requires Exclusive locks (X or IX).

Cursor controlled This type includes UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF. It also requires Exclusive locks (X or IX).

A statement that inserts, updates or deletes against a target table, based on the result from a sub-select statement, does two types of processing. The locks for the tables returned in the sub-select are determined by the rules for read-only processing; for the target table, by the rules for change processing.

Access Paths

An *access path* is the method selected by the optimizer for retrieving data from a specific table reference. The access path chosen by the optimizer can have a significant effect on the lock modes. For example, when an index scan is used to locate a specific row, the optimizer will likely choose row-level locking (IS) for the table. This type of access would be used to select information for a single employee from the EMPLOYEE table, that has an index on employee number (EMPNO), with a statement such as the following:

```
SELECT *
  FROM EMPLOYEE
  WHERE EMPNO = '000310';
```

Similarly, when no index is used, the entire table must be scanned in sequence to find the selected rows, and may acquire a single table level lock (S). For example, this type of access might be used to select all the male employees, using a statement such as this:

```
SELECT *
  FROM EMPLOYEE
  WHERE SEX = 'M';
```

The following tables provide an overview of which locks are obtained for what kind of access plan. See “Application Processing” on page 147 for definitions of the column headings. For definitions of the access methods, see the *Administration Guide*. Note that *cursor controlled* type processing uses the lock mode of the underlying cursor until the application finds a row to update or delete. For this type of processing, no matter what the lock mode of a cursor, an exclusive lock will always be obtained to perform the update or delete.

In the following tables, if only one lock mode is shown, it is a table level lock mode. If two lock modes are shown, the first is the table level lock mode and the second is the row level lock mode.

Table 7. Lock Modes for Table Scans

Isolation Level	Read-only	Intent to Change	Change
Access Method: ***Table scan with no predicates***			
RR	S	U	X
RS	IS / NS	IX / U	IX / X
CS	IS / NS	IX / U	IX / X
UR	IN	IX / U	IX / X
Access Method: ***Table Scan with predicates***			
RR	S	U	U
RS	IS / NS	IX / U	IX / U
CS	IS / NS	IX / U	IX / U
UR	IN	IX / U	IX / U

Table 8 (Page 1 of 2). Lock Modes for Index Scans

Isolation Level	Read-only	Intent to Change	Change
Access Method: ***Index Scan with no predicates***			
RR	S	IX / U	X
RS	IS / NS	IX / U	IX / X
CS	IS / NS	IX / U	IX / X
UR	IN	IX / U	IX / X
Access Method: ***Index Scan a single qualifying row***			
RR	IS / S	IX / U	IX / X
RS	IS / NS	IX / U	IX / X
CS	IS / NS	IX / U	IX / X

Table 8 (Page 2 of 2). Lock Modes for Index Scans

Isolation Level	Read-only	Intent to Change	Change
UR	IN	IX / U	IX / X
Access Method: ***Index Scan with start and stop predicates only***			
RR	IS / S	IX / S	IX / X
RS	IS / NS	IX / U	IX / X
CS	IS / NS	IX / U	IX / X
UR	IN	IX / U	IX / X
Access Method: ***Index Scan with predicates***			
RR	IS / S	IX / S	IX / U
RS	IS / NS	IX / U	IX / U
CS	IS / NS	IX / U	IX / U
UR	IN	IX / U	IX / U

Table 9 shows the lock modes for cases in which reading of the data pages is deferred to allow the list of rows to be:

- Further qualified using multiple indexes.
- Sorted for efficient prefetching.

The deferred access of the data pages implies that access to the row occurs in two steps and this results in more complex locking scenarios. There are two major categories which depend on the isolation level. Since the repeatable read isolation level keeps all locks acquired until the end of the transaction, the locks acquired in the first step are held and there is no need to acquire further locks in the second step. For the read stability and cursor stability isolation levels, locks must be acquired during the second step. To maximize concurrency, we don't acquire locks during the first step and rely on the re-application of all predicates to ensure that only qualifying rows are returned.

Table 9 (Page 1 of 2). Lock Modes for Index Scans used for Deferred Data Page Access

Isolation Level	Read-only	Intent to Change	Change
Access Method: ***Index Scan with no predicates***			
RR	IS / S	IX / S	X
RS	IN	IN	IN
CS	IN	IN	IN
UR	IN	IN	IN
Access Method: ***Deferred Data Page Access, after an index scan with no predicates***			
RR	IN	IX / S	X
RS	IS / NS	IX / U	IX / X
CS	IS / NS	IX / U	IX / X
UR	IN	IX / U	IX / X

Table 9 (Page 2 of 2). Lock Modes for Index Scans used for Deferred Data Page Access

Isolation Level	Read-only	Intent to Change	Change
Access Method: ***Index Scan with predicates***			
RR	IS / S	IX / S	IX / S
RS	IN	IN	IN
CS	IN	IN	IN
UR	IN	IN	IN
Access Method: ***Index Scan with start and stop predicates only***			
RR	IS / S	IX / S	IX / X
RS	IN	IN	IN
CS	IN	IN	IN
UR	IN	IN	IN
Access Method: ***Deferred Data Page Access, after an index scan with predicates***			
RR	IN	IX / S	IX / S
RS	IS / NS	IX / U	IX / U
CS	IS / NS	IX / U	IX / U
UR	IN	IX / U	IX / U

The access path is not controlled by the user; it is chosen by the Optimizer.

The access path used can affect the mode and granularity of a lock. For example, in an application using the repeatable read (RR) isolation level, an UPDATE query that uses a table scan without predicates, would use an X lock on the table. If rows were located through an index, the database manager might choose to lock individual rows of the table.

LOCK TABLE Statement

You can override the rules for acquiring initial lock modes by using the LOCK TABLE statement in an application.

The statement locks an entire table. Only the table specified in the LOCK TABLE statement is locked. Parent and dependent tables of the specified table are not locked. You must determine whether locking other tables that can be accessed is necessary to achieve the desired result in terms of concurrency and performance. The lock is not released until the unit of work is committed or rolled back.

If a table is normally shared among several users, you might want to lock it for the following reasons:

LOCK TABLE IN SHARE MODE

You want to access data that is *consistent in time*; that is, data current for a table at a specific point in time. If the table experiences frequent activity, the only way to ensure that the entire table remains stable is to lock it. For example, your application wants to take a snapshot of a table. However, during the time your application needs to process some rows of a table, other applications are updating rows you have not yet processed. This is allowed with repeatable read, but this action is not what you want.

As an alternative, your application can issue the LOCK TABLE IN SHARE MODE statement: no rows can be changed, regardless of whether you have retrieved them or not. You can then retrieve as many rows as you need, knowing that the rows you have retrieved have not been changed just before you retrieved them.

With LOCK TABLE IN SHARE MODE, other users can retrieve data from the table, but they cannot update, delete, or insert rows into the table.

LOCK TABLE IN EXCLUSIVE MODE

You want to update a large part of the table. It is less expensive and more efficient to prevent all other users from accessing the table than it is to lock each row as it is updated, and then unlock the row later when all changes are committed.

With LOCK TABLE IN EXCLUSIVE MODE, all other users are locked out; no other applications can access the table unless they are uncommitted read applications.

For more details on the LOCK TABLE statement, refer to the *SQL Reference* manual.

CLOSE CURSOR WITH RELEASE

When you close a cursor with the CLOSE CURSOR statement that includes the WITH RELEASE clause, all read locks (if any) that have been held for the cursor are released. Read locks are IS, S, and U table locks as well as S, NS, and U row locks. For more information on lock modes, see “Attributes of Locks” on page 140.

The WITH RELEASE clause has no effect for cursors that are operating under the CS or UR isolation levels. When specified for cursors that are operating under the RS or RR isolation levels, the WITH RELEASE clause ends some of the guarantees of those isolation levels. Specifically, an RS cursor may experience the *nonrepeatable read* phenomenon, and an RR cursor may experience either the *nonrepeatable read* or *phantom read* phenomenon.

If a cursor that is originally RR or RS is reopened after being closed using the WITH RELEASE clause, then new read locks will be acquired.

Summary of Locking Considerations

The following are points to remember about locking:

- Small units of work (frequent COMMIT statements) promote concurrent access of data by many users. Include COMMIT statements when your application is logically at a point of consistency; that is, when the data you have changed is consistent. When a COMMIT is issued, locks are released (except for table locks associated with cursors declared WITH HOLD).

- Locks are acquired even if your application merely reads rows, so it is still important to commit read-only units of work. This is because shared locks are acquired by repeatable read, read stability, and cursor stability isolation levels in read-only applications. With repeatable read and read stability, all locks are held until a COMMIT is issued, preventing other processes from updating the locked data, unless you close your cursor using the WITH RELEASE clause. In addition, catalog locks are acquired even in uncommitted read applications using dynamic SQL.
- The database manager ensures that your application does not retrieve uncommitted data (rows that have been updated by other applications but are not yet committed) unless you are using the uncommitted read isolation level.
- You can lock the entire table that you want to protect by issuing a LOCK TABLE statement:
  - To allow other applications to retrieve, but not update, delete, or insert rows
  - To prevent other applications (other than those with an uncommitted read isolation level) from accessing the rows of a table.
- When you close a cursor with the CLOSE CURSOR statement that includes the WITH RELEASE clause, all read locks (if any) that have been held for the cursor are released.
- When changing the configuration parameters affecting locking in a partitioned database, ensure that the changes are made to all of the partitions in the database.

Row Blocking

Row blocking is a technique that reduces database manager overhead by retrieving a *block* of rows in a single operation. These rows are stored in a cache, and each FETCH request in the application gets the next row from the cache. When all the rows in a block have been processed, another block of rows is retrieved by the database manager.

The cache is allocated when an application issues an OPEN CURSOR request and is deallocated when the cursor is closed. The size of the cache is determined by a configuration parameter which is used to allocate memory for the I/O block. The parameter used depends on whether the client is local or remote:

- For *local applications*, the parameter *aslheapsz* is used to allocate the cache for row blocking.
- For *remote applications*, the parameter *rqrioblk* on the client workstation is used to allocate the *cache* for row blocking. The cache is allocated on the database client.

For *local* applications, you can use the following formula to estimate how many rows are returned per block, where:

- *aslheapsz* is in pages of memory
- 4096 is the number of bytes per page
- *orl* is the output row length in bytes:

```
Rows per block = aslheapsz * 4096 / orl
```

For *remote* applications, you can use the following formula to estimate how many rows are returned per block, where:

- *rqrioblk* is in bytes of memory
- *orl* is the output row length in bytes:

```
Rows per block = rqrioblk / orl
```

Note that if you use the OPTIMIZE FOR *n* ROWS clause in a SELECT statement, the number of rows per block will be the minimum of the following:

- The value calculated in the above formula
- The value of *n* in the OPTIMIZE FOR clause

See the *Administration Guide* for information about these configuration parameters.

Use the BLOCKING option on the PREP and BIND commands to specify one of the following types of row blocking:

UNAMBIG Blocking occurs for read-only cursors and cursors not specified as “FOR UPDATE OF.” Ambiguous cursors are treated as updateable.

ALL Blocking occurs for read-only cursors and cursors not specified as “FOR UPDATE OF.” Ambiguous cursors are treated as read-only.

NO Blocking does not occur for any cursors. Ambiguous cursors are treated as read-only.

For details of these types of row blocking, refer to the PREP and BIND command descriptions in the *Command Reference* manual.

If no option is specified on the PREP and BIND commands, the default row blocking type is UNAMBIG. For the command line processor and call level interface, the default row blocking type is ALL.

See “Declaring and Using the Cursor” on page 55 for information about ambiguous and read-only cursors.

Adjusting the Optimization Class

When an SQL query is compiled, a number of optimization techniques can be used to determine the most efficient access plan for that query. Using more optimization techniques results in:

1. Improvements in run-time performance
2. Increased query compilation time
3. Increased system resource usage.

For this reason, you may wish to limit the number of techniques applied to optimizing your query by setting the optimization class. This can be particularly useful if you have:

- Very small databases or very simple dynamic queries

- Limited memory available at compile time on your database server
- A desire to reduce the query compilation (for example, PREPARE) time.

You may select from any of the query optimization classes described below, although class 0 and class 9 should be used only in special circumstances. Class 5 is the default. Classes 0, 1, and 2 use the Greedy join enumeration algorithm; for complex queries this algorithm considers far fewer alternative plans, and incurs significantly less compilation time, than classes 3 and above. Classes 3 and above use the Dynamic Programming join enumeration algorithm; this algorithm considers far more alternative plans, and can incur significantly more compilation time, than classes 0, 1, and 2 as the number of tables increases.

0 - This class directs the optimizer to use a minimal amount of optimization to generate an access plan. For example:

- Any non-uniform distribution statistics are not considered by the optimizer.
- Only basic query rewrite rules are applied (see *Administration Guide* for information about query rewrite).
- Greedy join enumeration occurs (see *Administration Guide*).
- Only nested loop join and index scan access methods are enabled (see *Administration Guide*).
- List prefetch and index ANDing are disabled as access methods.
- The star join strategy is not considered.

This class should only be used in special circumstances requiring the lowest possible query compilation overhead. An application consisting entirely of very simple dynamic SQL statements which access well-indexed tables is a good example of where query optimization class 0 is appropriate.

1 - This class directs the optimizer to use a degree of optimization which is roughly comparable to DB2/6000 Version 1, plus some additional low cost features not found in Version 1. In particular:

- Any non-uniform distribution statistics are not considered by the optimizer.
- Only a subset of the query rewrite rules are applied, including those provided in DB2/6000 Version 1.
- Greedy join enumeration (see *Administration Guide*.)
- List prefetch and index ANDing are disabled as access methods.

Optimization class 1 is quite similar to class 0 except that Merge Scan joins and table scans are also available. For definitions of Merge Scan joins and Nested Loop joins, see the *Administration Guide*.

2 - This class directs the optimizer to use a degree of optimization which significantly improves upon that of class 1, while keeping the compilation cost significantly lower than classes 3 and above for complex queries. In particular:

- All available statistics, including both frequency and quantile non-uniform distribution statistics, are utilized.
- All of the query rewrite rules are applied, except computationally intensive rules which are applicable only in very rare cases.
- Greedy join enumeration (see *Administration Guide*) is used.

- A wide range of access methods are considered, including list prefetch.
- The star join strategy is considered, if applicable.

Optimization class 2 is quite similar to class 5 except that it uses Greedy join enumeration rather than Dynamic Programming. This class has the most optimization of all the optimization classes that use the Greedy join enumeration algorithm, which considers fewer alternatives for complex queries, and therefore consumes less compilation time than classes 3 and above. It is therefore recommended for very complex queries in a decision support or on-line analytic processing (OLAP) environment. In such cases, there is a good chance the same query is executed infrequently, so that its access plan is unlikely to remain in the cache until the next occurence of the query.

3 - This class requests that a moderate amount of optimization be performed to generate an access plan. This class comes closest to matching the query optimization characteristics of DB2 for MVS/ESA or OS/390. This optimization class has the following characteristics:

- Non-uniform distribution statistics, which track frequently occurring values are used, if available.
- Most query rewrite rules, including subquery-to-join transformations are applied.
- Dynamic programming join enumeration (see *Administration Guide*):
  - Limited use of composite inner tables (see *Administration Guide*)
  - Limited use of Cartesian products for star schemas involving “look-up” tables (see *Administration Guide*)
- A wide range of access methods are considered, including list prefetch and index ANDing.

This class is suitable for a broad range of applications. Using this class gives the optimizer a better chance of selecting an excellent access plan for queries with four or more joins. However, the optimizer might fail to consider a better plan which would be chosen with the default query optimization class.

5 - This class directs the optimizer to use a significant amount of optimization to generate an access plan. For example, class 5 has the following characteristics:

- All available statistics including both frequency and quantile non-uniform distribution statistics.
- All of the query rewrite rules are applied, except computationally intensive rules which are applicable only in very rare cases.
- Dynamic programming join enumeration (see *Administration Guide*):
  - Limited use of composite inner tables (see *Administration Guide*)
  - Limited use of Cartesian products for star schemas involving “look-up” tables (see *Administration Guide*)
- A wide range of access methods are considered, including list prefetch and index ANDing.

When the optimizer detects that the additional resources and processing time are not warranted for complex dynamic SQL queries, optimization is reduced. The extent or size of the reduction is dependent on the machine size and the number of predicates.

When the query optimizer reduces the amount of query optimization performed, it continues to apply all the query rewrite rules that would normally be applied. However, it does use the greedy join enumeration method and reduces the number of access plan combinations that are considered.

Query optimization class 5 is an excellent choice for a mixed environment consisting of both transactions and complex queries. This optimization class has been designed to apply the most valuable query transformations and other query optimization techniques in an efficient manner.

7 - This class directs the optimizer to use a significant amount of optimization to generate an access plan. It is the same as query optimization class 5 except that it does not reduce the amount of query optimization for complex dynamic SQL queries.

9 - This class directs the optimizer to use all available optimization techniques. These include:

- All available statistics
- All query rewrite rules
- All possibilities for join enumerations, including Cartesian products and unlimited composite inners
- All access methods.

This class can greatly expand the number of possible access plans that are considered by the optimizer. This class should be used to determine whether more comprehensive optimization can generate a better access plan for very complex and very long-running queries using large tables. Explain and performance measurements should be used to verify that a better plan has been found.

How Do You Set the Optimization Class?

The way to request a specific query optimization class depends on whether you are using static or dynamic SQL.

- *Static SQL statements* use the optimization class specified on the PREP and BIND commands. The QUERYOPT column in the SYSCAT.PACKAGES catalog table records the optimization class used to bind the package. If the package is rebound either implicitly or using the REBIND PACKAGE command, this same optimization class will be used for the static SQL statements. If you want to change the optimization class used for these static SQL statements, you must use the BIND command. If you do not specify the optimization class, DB2 uses the default optimization as specified by *dft_queryopt*.

- *Dynamic SQL statements* use the optimization class specified by the CURRENT QUERY OPTIMIZATION special register which is set using the SQL SET statement. For example, the following statement sets the optimization class to 1:

  ```
  SET CURRENT QUERY OPTIMIZATION = 1
  ```

  To ensure that a dynamic SQL statement always uses the same optimization class, you may want to include this SET statement in your application program. For more information, refer to the *SQL Reference*.

  If the CURRENT QUERY OPTIMIZATION register has not been set, dynamic statements will be bound using the default query optimization class. The default value for both dynamic and static SQL is determined by value of the configurable database parameter DFT_QUERYOPT. Class 5 is the default query optimization class unless you have changed the default. The default values for the bind option and the special register are taken from the DFT_QUERYOPT configuration parameter.

How Much Optimization is Necessary?

Most statements will be adequately optimized using a reasonable amount of resources with the default query optimization class. The query compilation time and resource consumption, at a given optimization class, is primarily influenced by the complexity of the query, particularly the number of joins and subqueries. However, compilation time and resource usage are also affected by the amount of optimization performed for the various optimization classes. For any optimization class, you can expect to see a greater difference in query compilation time and resource consumption for a very complex query than for a simple one.

The following may help you select which optimization class to use:

- Start by using the default query optimization class.
- If you wish to use a class other than the default, try class 1, 2 or 3 first.
- Use a low optimization class (0 or 1) for queries having very short run-times, that is, queries taking less than one second. (See the following discussion for additional criteria about when to choose a low optimization class.)
- Use optimization class 1 or 2 if you have many tables with many of the join predicates that are on the same column, and if compilation time is a concern.
- Use a higher optimization class (3, 5, or 7) for long running queries, that is, queries taking more than 30 seconds.
- Under normal circumstances, you should not use optimization class 9.
- For queries that run a long time, run the query using db2batch to determine how much of the time is spent in compilation and how much is spent in execution.
  - If most of the time is spent in compilation then reduce the optimization class.
  - If most of the time is spent in execution then consider a higher optimization class.

Note that query optimization classes 1, 2, 3, 5, and 7 are all suitable for general purpose use.

Only if you require further reductions in query compilation time and you know the kind of SQL (for example, extremely simple statements) that will be executed should you consider class 0. This SQL will tend to have the following characteristics:

- Access to a single or only a few tables
- Fetches a single or only a few rows
- Uses fully qualified, unique indexes.

Online transaction processing (OLTP) transactions are good examples of this kind of SQL.

Complex queries may require different amounts of optimization to select the best access plan. You may wish to consider using higher optimization classes for queries exhibiting the following characteristics:

- Access to large tables
- A large number of predicates
- Many subqueries
- Many joins
- Many set operators, such as UNION and INTERSECT
- Many qualifying rows
- GROUP BY and HAVING operations
- Nested table expressions
- A large number of views.

Decision support queries or month-end reporting queries against fully normalized databases are good examples of complex queries where at least the default query optimization class should be used.

Another reason to use higher query optimization classes is SQL which was produced by a query generator. Many query generators create SQL which is not efficient. Poorly written queries, including those produced by a query generator, may require additional optimization to make it possible to select a good access plan. Using query optimization class 2 and higher can improve poorly written SQL queries.

The use of static or dynamic SQL, and whether the same dynamic SQL is repeatedly executed are also important considerations. For static SQL, the query compilation time and resources are expended just once and the resulting plan can be used many times. In general, static SQL should always use the default query optimization class. Dynamic statements are bound and executed at run time; therefore, you should consider whether the overhead of additional optimization for dynamic statements improves your overall performance. However, if the same dynamic SQL statement is executed repeatedly, the selected access plan will be cached. For the purposes of selecting a query optimization class, the statement can be treated like a static SQL statement.

For more information, see “Characteristics and Reasons for Using Static SQL” on page 39 and “Why Use Dynamic SQL?” on page 91.

If you think you have a query that could benefit from additional optimization, but you are not sure, or you are concerned about compilation time and resource usage, you may want to perform some benchmark testing. This testing can help you quantify the benefits obtained from different optimization classes. Refer to the *Administration Guide* for general techniques and the specific use of the `db2batch` tool. When designing and running your benchmark test, consider whether the SQL statements in your application are static or dynamic:

- For **dynamic** SQL statements, your testing should compare the average run time for the statement. You can use the following formula to help you calculate the average run time:

  ```
  compile time + sum of execution times for all iterations
  --------------------------------------------------------
                 number of iterations
  ```

  where, the `number of iterations` represents the number of times that you expect that the SQL statement will be executed each time it is compiled.

  Note: Following the initial compilation, dynamic SQL statements are recompiled when a change to the environment requires the statement to be recompiled. Once cached, a SQL statement does not need to be compiled again since subsequent PREPARE statements will re-use the cached statement assuming the environment does not change. Refer to the *Administration Guide* for information about a cache that can improve performance when working with dynamic SQL statements.

- For **static** SQL statements, your testing should compare the statement run times.

  Note: While you may also be interested in the compile time of static SQL, the total (compile and run) time for the statement is difficult to use in any meaningful context. Comparing the total time does not recognize the fact that a static SQL statement can be run many times for each time it is bound and that it is generally not bound during run time.

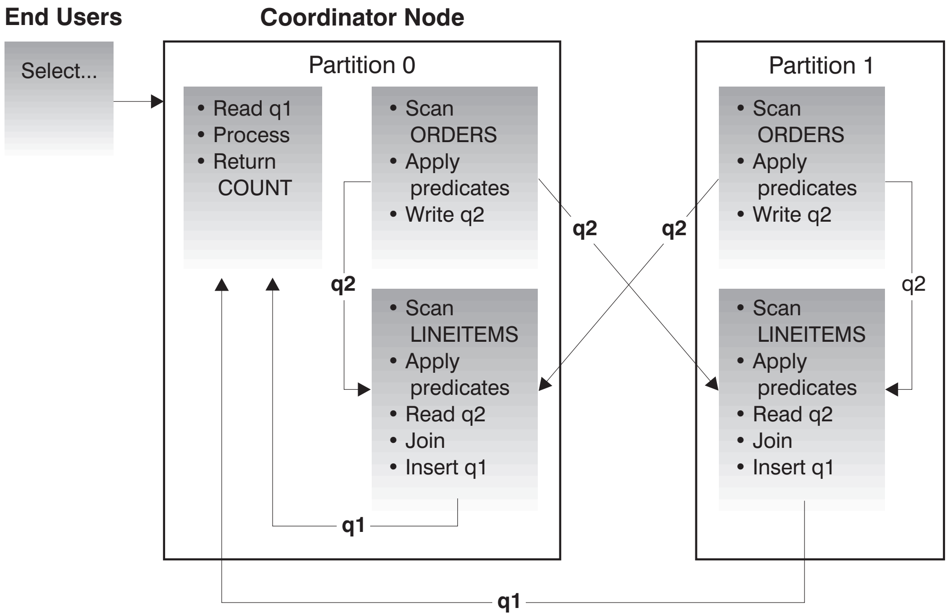
Join Strategies in a Partitioned Database

The following sections describe the join strategies that are possible in a partitioned database environment. The DB2 optimizer automatically selects the best join strategy depending on the requirements of each application. The join strategies are presented here to help you understand what is happening in each strategy. In the descriptions that follow, a *directed* table queue is one whose rows are hashed to one of the receiving database partitions. A *broadcast* table queue is one whose rows are sent to all of the receiving database partitions (that is, it is not hashed). In the diagrams for this section q1, q2, and q3 refer to table queues in the examples. Also the tables that are referenced are divided across two database partitions for the purpose of these scenarios. The arrows indicate the direction in which the table queues are sent. The coordinator node is partition 0.

For information on join dependencies, see the *SQL Reference* manual.

Broadcast Outer-Table Joins

This parallel join strategy can be used if there are no equijoin predicates between the joined tables. It can also be used in other situations in which it is the most cost-effective join method. In this situation, one of the tables is broadcast to all the database partitions of the outer joined table. An example is shown in Figure 12.

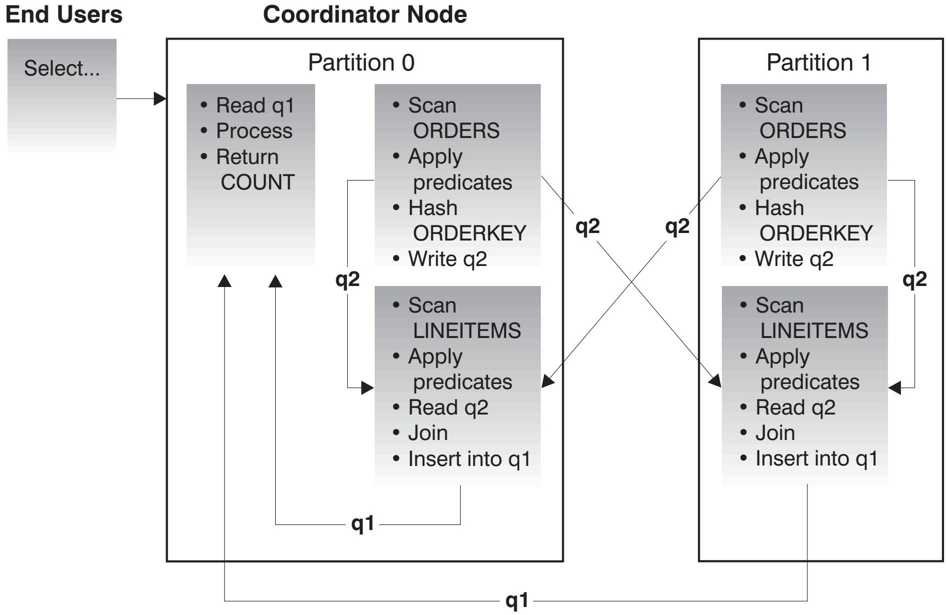


The ORDERS table is sent to all database partitions that have the LINEITEM table.
Table queue q2 is broadcast to all database partitions of the inner table.

Figure 12. Broadcast Outer-Table Join Example

Directed Outer-Table Joins

In this join strategy, each row of the outer table is sent to one database partition of the inner table (based on the partitioning attributes of the inner table). The join occurs on this database partition. An example is shown in Figure 13 on page 161.

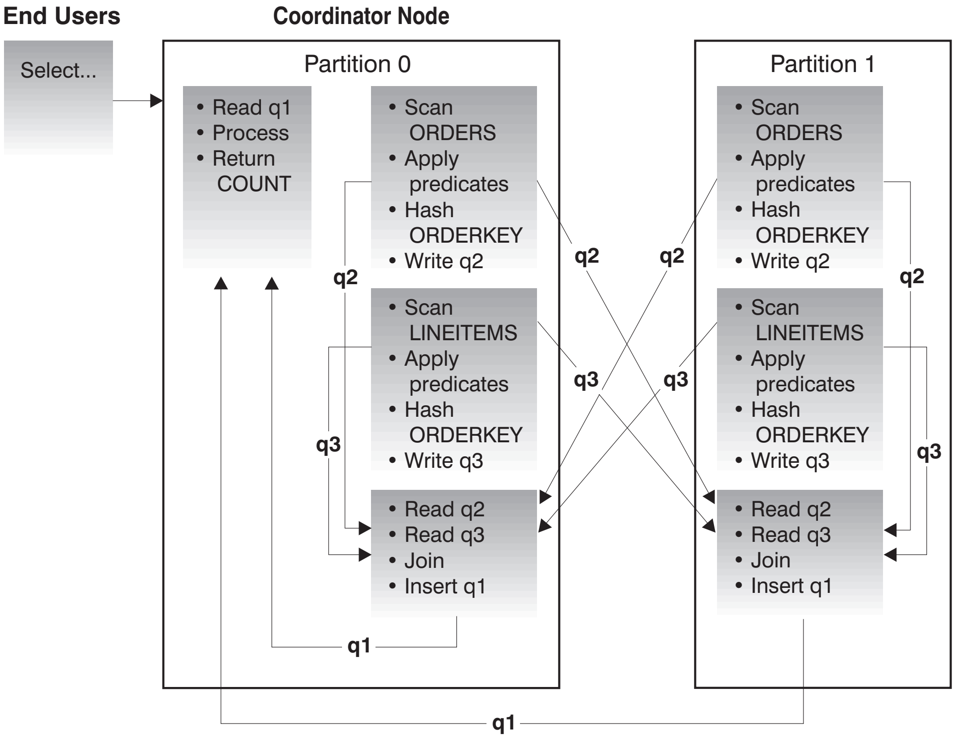


The LINEITEM table is partitioned on the ORDERKEY column.
The ORDERS table is partitioned on a different column.
The ORDERS table is hashed and sent to the correct LINEITEM table database partition.
In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEMS.ORDERKEY.

Figure 13. Directed Outer-Table Join Example

Directed Inner-Table and Outer-Table Joins

With this strategy, rows of the outer and inner tables are directed to a set of database partitions, based on the values of the joining columns. The join occurs on these database partitions. An example is shown in Figure 14 on page 162.



Neither table is partitioned on the ORDERKEY column.

Both tables are hashed and are sent to new database partitions where they are joined.

Both table queue q2 and q3 are directed.

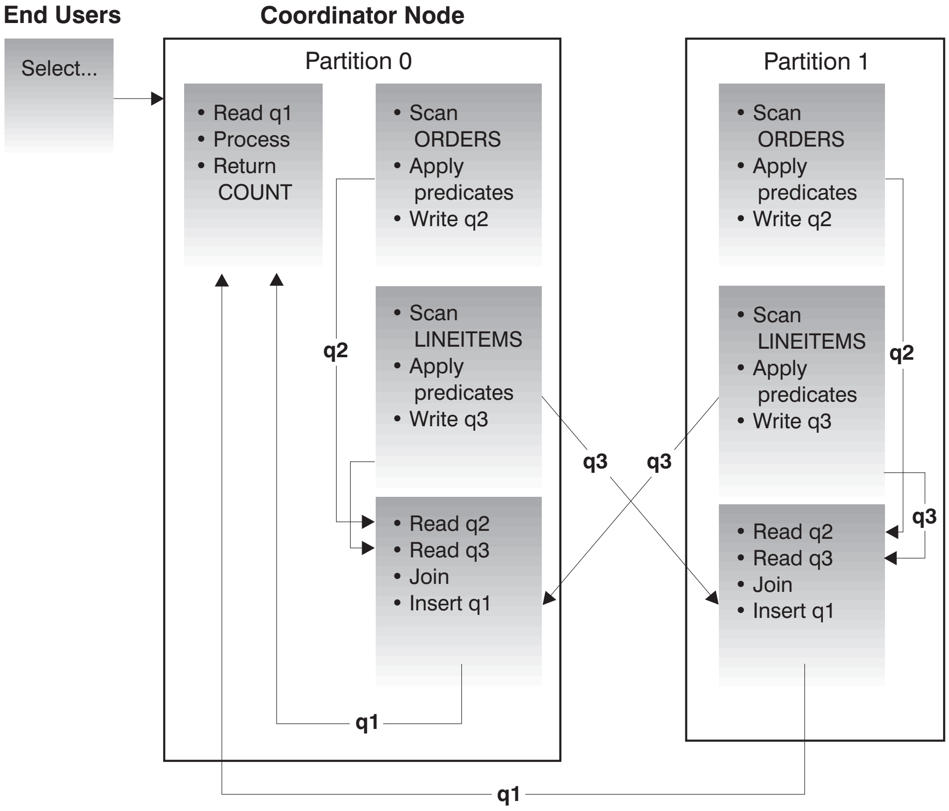
In this example, the join predicate is assumed to be:

ORDERS.ORDERKEY = LINEITEMS.ORDERKEY

Figure 14. Directed Inner-Table and Outer-Table Join Example

Broadcast Inner-Table Joins

With this strategy, the inner table is broadcast to all the database partitions of the outer join table. An example is shown in Figure 15 on page 163.

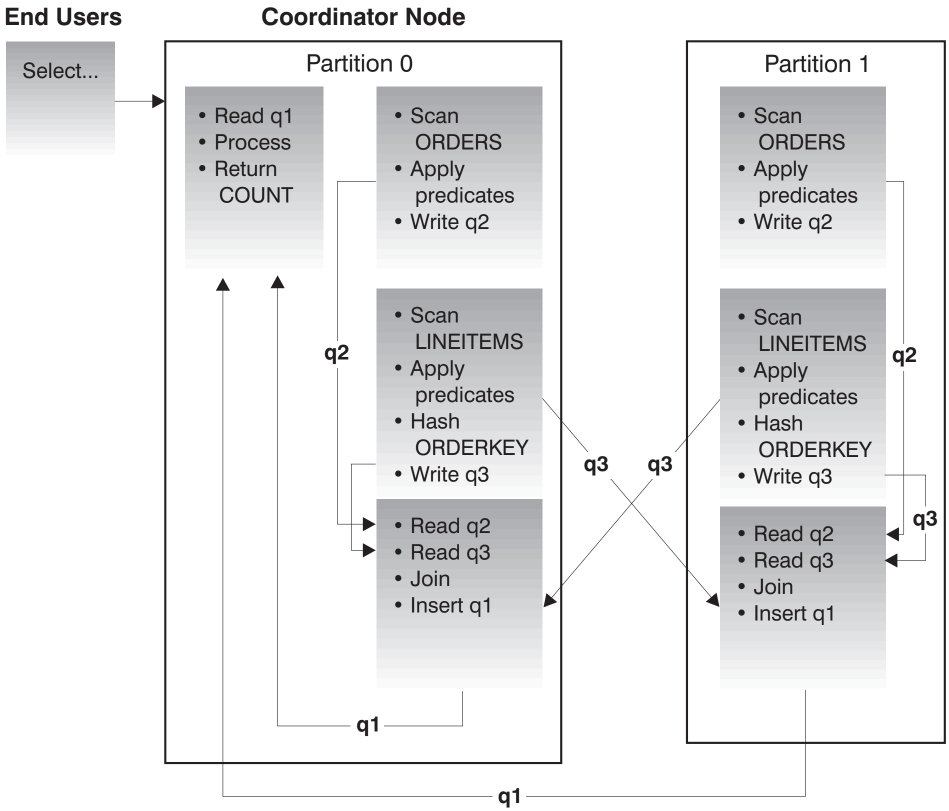


The ORDERS table is sent to all database partitions that have the LINEITEM table. Table queue q2 is broadcast to all database partitions of the inner table.

Figure 15. Broadcast Inner-Table Join Example

Directed Inner-Table Joins

With this strategy, each row of the inner table is sent to one database partition of the outer join table (based on the partitioning attributes of the outer table). The join occurs on this database partition. An example is shown in Figure 16 on page 164.

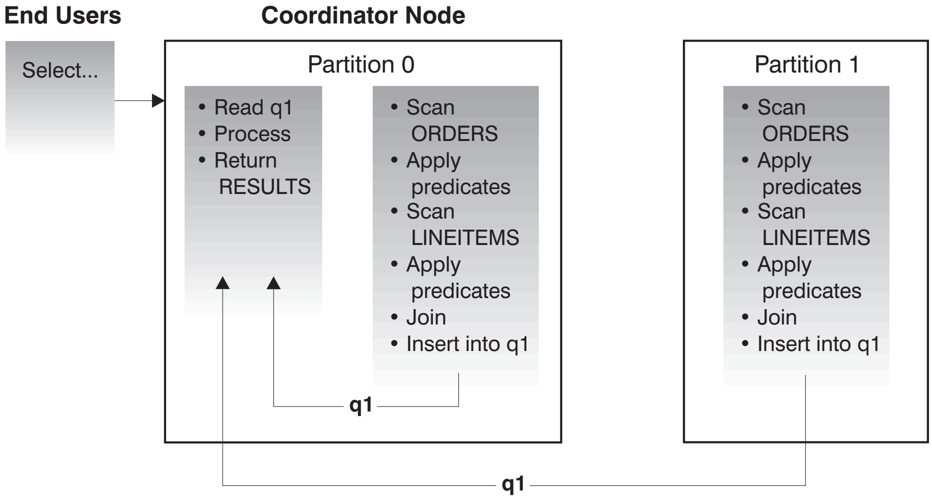


The ORDERS table is partitioned on the ORDERKEY column.
The LINEITEM table is partitioned on a different column.
The LINEITEM table is hashed and sent to the correct ORDERS table database partition.
In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEMS.ORDERKEY.

Figure 16. Directed Inner-Table Join Example

Collocated Joins

For the optimizer to consider a collocated join, the joined tables must be collocated, and all pairs of the corresponding partitioning key must participate in the equijoin predicates. An example is shown in Figure 17 on page 165.



Both the LINEITEM and ORDERS tables are partitioned on the ORDERKEY column. The join is done locally at each database partition. In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEM.ORDERKEY.

Figure 17. Collocated Join Example

Table Queues

A table queue is used to pass table data from one database partition to another. Each table queue is used to pass the data in a single direction.

The compiler decides where table queues are required, and includes them in the plan. When the plan is executed, the connections between the database partitions initiate the table queues. The table queues close as processes end.

There are two types of table queues:

- `Asynchronous table queues`. These table queues are known as asynchronous because they read rows in advance of any FETCH being issued by the application. When the FETCH is issued, the row is retrieved from the table queue.

  Asynchronous table queues are used when you specify the FOR FETCH ONLY clause on the SELECT statement. If you are only fetching rows, the asynchronous table queue is faster.

- `Synchronous table queues`. These table queues are known as synchronous because they read one row for each FETCH that is issued by the application. At each database partition, the cursor is positioned on the next row to be read from that database partition.

  Synchronous table queues are used when you do not specify the FOR FETCH ONLY clause on the SELECT statement. In a partitioned database environment, if you are updating rows, the database manager will use the synchronous table queues.

Chapter 5. Writing Stored Procedures

This chapter describes how stored procedures can be used for database manager applications running in a client/server environment.

This technique allows an application running on a client to call a procedure stored on a database server. This *stored procedure* executes and accesses the database locally and returns information to the client application.

To use this technique, an application must be written in two separate procedures. The calling procedure is contained in a client application and executes on the client. The *stored procedure* executes at the location of the database on the database server.

The following topics are discussed:

- Why Use Stored Procedures?
- Invoking Stored Procedures
- Writing Stored Procedures on DB2
- Building the Stored Procedure Application
- Resolving Problems
- Working with Not-Fenced Stored Procedures
- Example Output-SQLDA Programs
- Example Input-SQLDA Programs
- Registering Stored Procedures
- Returning Result Sets From Stored Procedures

Why Use Stored Procedures?

Figure 18 shows how a normal database manager application accesses a database located on a database server.

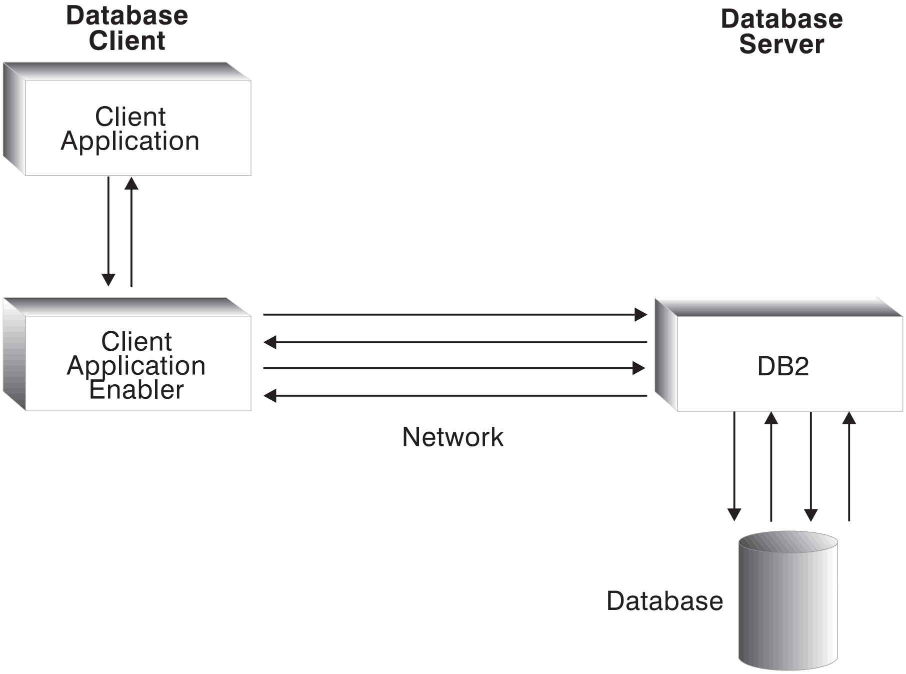


Figure 18. Application Accessing a Database on a Server

All database access must go across the network which, in some cases, results in poor performance.

Using stored procedures allows a client application to pass control to a stored procedure on the database server. This allows the stored procedure to perform intermediate processing on the database server, *without transmitting unnecessary data across the network*. Only those records that are actually required at the client need to be transmitted. This can result in reduced network traffic and better overall performance. Figure 19 shows this feature.

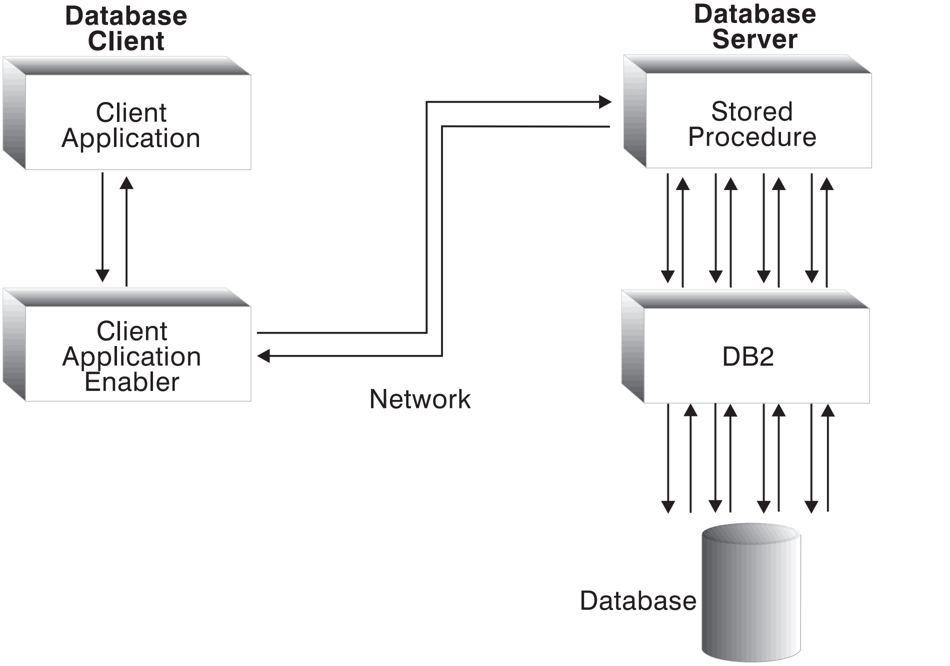


Figure 19. Application Using a Stored Procedure

Applications using stored procedures have the following advantages:

- Reduced network traffic.

  Applications that process large amounts of data but require only a subset of the data to be returned to the user. A properly designed application using stored procedures returns only the data that is needed by the client; the amount of data transmitted across the network is reduced.

- Improved performance of server intensive work.

  Applications executing SQL statements that can be grouped together without user intervention. The more SQL statements that are grouped together, the larger the savings in network traffic. A typical application requires two trips across the network for each SQL statement, whereas an application using the stored procedure technique requires two trips across the network for each *group* of SQL statements. This reduces the number of trips, resulting in a savings from the overhead associated with each trip.

- Access to features that exist only on the database server.

  These features include:

  – Commands to list directories on the server (such as LIST DATABASE DIRECTORY and LIST NODE DIRECTORY) can only run on the server.
  – The stored procedure may have the advantage of increased memory and disk space if the server computer is so equipped.
  – Additional software installed only on the database server could be accessed by the stored procedure.

Invoking Stored Procedures

You can invoke a stored procedure stored at the location of the database by using the SQL CALL statement. See the *SQL Reference* for a complete description of the CALL statement. Using the CALL statement is the recommended method of invoking stored procedures. For considerations on writing stored procedures in REXX, see “REXX Stored Procedures” on page 503. For considerations on writing stored procedures in Java, see “Creating and Using Java Stored Procedures” on page 516.

Writing Stored Procedures on DB2

An application using stored procedures must be written in two separate procedures. The calling procedure is contained in a client application, and executes on the client. It can be written in any of the supported host languages. The stored procedure executes at the location of the database on the database server, and must be written in one of the supported languages for that database server.

The two procedures must be built in separate steps.

The client application performs the following:

1. Declares, allocates, and initializes storage for the optional data structures and host variables.
2. Connects to a database. (It does this by executing the CONNECT TO statement, or by doing an implicit connect. See the *SQL Reference* for details.)
3. Executes on the client.
4. Invokes the stored procedure through the SQL CALL statement.
5. Performs a COMMIT or ROLLBACK to the database.
6. The application disconnects from the database.

Note that you can code SQL statements in any of the above steps.

When invoked, the stored procedure performs the following:

1. Accepts the SQLDA data structure from the client application. (Host variables are passed through an SQLDA data structure generated by the database manager when the SQL CALL statement is executed.)
2. Executes on the database server under the same transaction as the client application.
3. Returns SQLCA information and optional output data to the client application.

The stored procedure executes when called by the client application. Control is returned to the client when the server procedure finishes processing. You can take several stored procedures and put them into one library.

Client Application

The client application performs several steps before calling the stored procedure. It must be connected to a database, and it must declare, allocate, and initialize the SQLDA structure or host variables. The SQL CALL statement can accept a series of host variables, or an SQLDA structure. (See the *SQL Reference* for a description of the SQL CALL statement.)

Note: Do not allocate storage for these structures on the database server. The database manager automatically allocates duplicate storage based upon the storage allocated by the client application. Do not alter any storage pointers for the input/output parameters on the stored procedure side. Attempting to replace a pointer with a locally created storage pointer will cause an error with SQLCODE -1133 (SQLSTATE 39502).

SQLDA Structure

For the SQLDA structure used, perform the following steps before calling the stored procedure:

1. Allocate storage for the structure with the desired number of base SQLVAR elements.
2. Set the SQLN field to the number of SQLVAR elements allocated.
3. Set the SQLD field to the number of SQLVAR elements actually used.
4. Initialize each SQLVAR element used as follows:
   - Set the SQLTYPE field to the proper data type.
   - Set the SQLLEN field to the size of the data type.
   - Allocate storage for the SQLDATA and SQLIND fields based upon the values in SQLTYPE and SQLLEN.

If your application will be working with character strings defined as FOR BIT DATA, you will need to initialize the SQLDAID field to indicate that the SQLDA includes FOR BIT DATA definitions and the SQLNAME field of each SQLVAR that defines a FOR BIT DATA element. If your application will be working with large objects, that is, data with types of CLOB, BLOB, or DBCLOB, you will also need to initialize the secondary SQLVAR elements. See the information on the SQLDA structure in the *SQL Reference*.

Allocating Host Variables

Following are the steps to allocate the necessary input host variables on the client side of a stored procedure:

- Declare enough SQLVARs for all input variables that will be passed to the stored procedure
- Determine which input SQLVARs can also be used to return values back from the stored procedure to the client

- Declare SQLVARs for any additional values returned from the stored procedure to the client

When writing the client portion of your stored procedure, you should attempt to overload as many of the SQLVARs as possible in order to increase the efficiency of handling SQLVARs. For example, when returning an SQLCODE to the client from the stored procedure, try to use an input SQLVAR that is declared as an INTEGER to return the SQLCODE. Unless you need to reuse the values in the SQLDA on the client, there is no problem overwriting the content of any SQLDA buffer.

In addition, the client application should set the indicator of output-only SQLVARs to -1 as discussed in “Data Structure Manipulation” on page 173. This will improve the performance of the parameter passing mechanism by avoiding having to pass the contents of the SQLDATA pointer as only the indicator is sent. You should set the SQLTYPE field to a nullable data type for these parameters. If the SQLTYPE indicates a non-nullable data type, the indicator variable is not checked by the database manager.

Running the Client Application

The client application must ensure that a database connection has been made before invoking the stored procedure, or an error is returned. After the database connection and data structure initialization, the client application calls the stored procedure and passes any required data. The application disconnects from the database. Note that you can code SQL statements in any of the above steps.

Stored Procedure

The stored procedure is invoked by the SQL CALL statement and executes using data passed to it by the client application. Information is returned to the client application using the stored procedure’s SQLDA structure.

The parameters of the SQL CALL statement are treated as both input and output parameters and are converted into the following format for the stored procedure:

```
SQL_API_RC SQL_API_FN proc_name( void *reserved1,
                                 void *reserved2,
                                 struct sqlda *inoutsqlda,
                                 struct sqlca *sqlca )
```

The `SQL_API_FN` is a macro that specifies the calling convention for a function that may vary across each supported operating system. This macro is required when you write stored procedures or UDFs.

Following is an example of how a CALL statement maps to a server’s parameter list:

```
CALL OUTSRV (:empno:empind,:salary:salind)
```

The parameters to this call are converted into an SQLDA structure with two SQLVARs. The first SQLVAR points to the `empno` host variable and the `empind` indicator variable. The second SQLVAR points to the `salary` host variable and the `salind` indicator variable.

Notes:

1. The SQLDA structure is not passed to the stored procedure if the number of elements, SQLD, is set to 0. In this case, if the SQLDA is not passed, the stored procedure receives a NULL pointer.
2. You cannot execute any connection-related statements or commands, such as, CONNECT, CONNECT TO, CONNECT RESET, CREATE DATABASE, DROP DATABASE, BACKUP, RESTORE, or FORWARD RECOVERY in a stored procedure.
3. Stored procedures run in the background, so you cannot write to the screen. However, you can write to a file.
4. Stored procedures cannot contain commands that would terminate the current process. A stored procedure should always return control to the client without terminating the current process.
5. The values of all environment variables beginning with 'DB2' are captured at the time the database manager is started with `db2start`, and are available in all stored procedures whether or not they are fenced. The only exception is the `DB2CKPTR` environment variable. Note that the environment variables are *captured*; any changes to the environment variables after `db2start` is issued are not passed to the stored procedures.
6. On UNIX-based systems, your stored procedure runs under the UID of the DB2 Agent Process (NOT FENCED), or the UID which owns the `db2dari` executable (FENCED). This UID controls the system resources available to the stored procedure. For information on the `db2dari` executable, see the *Quick Beginnings* book for your platform.

Data Structure Manipulation

The database manager automatically allocates a duplicate SQLDA structure at the database server. To reduce network traffic, it is important to indicate which host variables are input-only, and which ones are output-only. The client procedure should set the indicator of output-only SQLVARs to -1. The server procedure should set the indicator for input-only SQLVARs to -128. This allows the database manager to choose which SQLVARs are passed.

Note that an indicator variable is not reset if the client or the server sets it to a negative value (indicating that the SQLVAR should not be passed). If the host variable to which the SQLVAR refers is given a value in the stored procedure or the client code, its indicator variable should be set to zero or a positive value so that the value is passed. For example, consider a stored procedure which takes one output-only parameter, called as follows:

```
empind = -1;
EXEC SQL CALL storproc(:empno:empind);
```

When the stored procedure sets the value for the first SQLVAR, it should also set the value of the indicator to a non-negative value so that the result is passed back to `empno`.

Input/Output SQLDA and SQLCA Structures: The stored procedure runs using any information passed in the input variables of the SQLDA structure. Information is returned to the client in the output variables of the SQLDA. Do not change the value of the SQLD, SQLTYPE, and SQLLEN fields of the SQLDA, as these fields are compared to the original values set by the client application before data is returned. If they are different, one of the following SQLCODEs is returned:

SQLCODE -1113 (SQLSTATE 39502)	The data type of a variable (that is, the value in SQLTYPE) has changed.
SQLCODE -1114 (SQLSTATE 39502)	The length of a variable (that is, the value in SQLLEN) has changed.
SQLCODE -1115 (SQLSTATE 39502)	The SQLD field has changed.

In addition, do not change the pointer for the SQLDATA and the SQLIND fields, although you can change the value that is pointed to by these fields.

Note: It is possible to use the same variable for both input and output.

Before the stored procedure returns, SQLCA information should be explicitly copied to the SQLCA parameter of the stored procedure.

Return Values

The return value of the stored procedure is never returned to the client application. It is used by the database manager to determine if the server procedure should be released from memory upon exit.

The stored procedure returns with one of the following values:

SQLZ_DISCONNECT_PROC Tells the database manager to release (unload) the library.

SQLZ_HOLD_PROC Tells the database manager to keep the server library in main memory so that the library will be ready for the next invocation of the stored procedure. This may lead to better performance.

If the stored procedure is invoked only once, `SQLZ_DISCONNECT_PROC` should be returned.

If the client application issues multiple calls to invoke the same stored procedure, `SQLZ_HOLD_PROC` should be the return value of the stored procedure. The stored procedure will not be unloaded.

If `SQLZ_HOLD_PROC` is used, the last invocation of the server request should return the value `SQLZ_DISCONNECT_PROC` to free the server library from main memory. Otherwise, the library remains in main memory until the database manager is stopped. As an alert to the server, the client application could pass a flag in one of the parameters indicating the final call.

Code Page Considerations

The code page considerations depend on the server.

When a client program (using, for example, code page A) calls a remote stored procedure that accesses a database using a different code page (for example, code page Z), the following events occur:

1. Input character string parameters (whether defined as host variables or in an SQLDA in the client application) are converted from the application code page (A) to the one associated with the database (Z). Conversion does not occur for data defined in the SQLDA as FOR BIT DATA[4]
2. Once the input parameters are converted, the database manager does not perform any more code page conversions.

   Therefore, **you must run the stored procedure using the same code page as the database**, in this example, code page Z. It is a good practice to prep, compile, and bind the server procedure using the same code page as the database.
3. When the stored procedure finishes, the database manager converts the output character string parameters (whether defined as host variables or in an SQLDA in the client application) and the SQLCA character fields from the database code page (Z) back to the application code page (A). Conversion does not occur for data defined in the SQLDA as FOR BIT DATA[4].

Note: If the server being accessed is running Version 1 of DB2 for OS/2, the client code page **must** be the same as the database code page.

For more information on this topic, see “Conversion Between Different Code Pages” on page 374.

C++ Consideration

When writing a stored procedure in C++, you may want to consider declaring the procedure name as:

```
extern "C" SQL_API_RC SQL_API_FN proc_name( reserved1, reserved2,
                                            inoutsqlda, sqlca )
```

The `extern "C"` prevents type decoration (or mangling) of the function name by the C++ compiler. Without this declaration, you have to include all the type decoration for the function name when you call the stored procedure.

Java Considerations

When creating a stored procedure in the Java language, you must use the CREATE PROCEDURE statement to register the procedure to the system catalog table SYSCAT.PROCEDURES. For details on writing stored procedures in Java, see “Java

[4] If the parameter of the stored procedure is defined as FOR BIT DATA at the server, conversion does not occur for a CALL statement to DB2 for MVS/ESA, DB2 for OS/390, or DB2 for AS/400, regardless of whether it is explicitly specified in the SQLDA. (Refer to the section on the SQLDA in the *SQL Reference* for details.)

UDFs and Stored Procedures" on page 509, or refer to the CREATE PROCEDURE statement in the *SQL Reference*.

Graphic Host Variable Considerations

Any stored procedure written in C or C++, that receives or returns graphic data through its parameter input or output SQLDA should generally be precompiled with the WCHARTYPE NOCONVERT option. This is because graphic data passed through these SQLDAs is considered to be in DBCS format, rather than the `wchar_t` process code format. Using NOCONVERT means that graphic data manipulated in SQL statements in the stored procedure will also be in DBCS format, matching the format of the parameter data.

With WCHARTYPE NOCONVERT, no character code conversion occurs between the graphic host variable and the database manager. The data in a graphic host variable is sent to, and received from, the database manager as unaltered DBCS characters. Note that if you do not use WCHARTYPE NOCONVERT, it is still possible for you to manipulate graphic data in `wchar_t` format in a stored procedure; however, you must perform the input and output conversions manually.

CONVERT can be used in fenced stored procedures, and it will affect the graphic data in SQL statements within the stored procedure, but not through the stored procedure's interface. Not-fenced stored procedures must be built using the NOCONVERT option.

In summary, graphic data passed to or returned from a stored procedure through its input or output SQLDA is in DBCS format, regardless of how it was precompiled with the WCHARTYPE option.

For important information on handling graphic data in C applications, see "Handling Graphic Host Variables" on page 440. For information on EUC code sets and application guidelines, refer to "Japanese and Traditional-Chinese EUC Code Set Considerations" on page 377, and more specifically to "Considerations for Stored Procedures" on page 381.

Distributed Unit of Work (DUOW) Consideration

Stored procedures that are called by applications in which the CONNECT TYPE 2 DUOW parameter is in effect, are restricted from issuing COMMIT or ROLLBACK, either dynamically or statically.

Summary of Data Structure Usage

Table 10 summarizes the use of the various structure fields by the stored procedures application. In the table, `sqlda` is an SQLDA structure passed to the stored procedure and `n` is a numeric value indicating a specific SQLVAR element of the SQLDA. The numbers on the right refer to the notes following the table.

Table 10. Stored Procedures Parameter Variables

Input/Output SQLDA	sqlda.SQLDAID				4					
	sqlda.SQLDABC				4					
	sqlda.SQLN		2		4					
	sqlda.SQLD		2	3		5				
Input/Output SQLVAR	sqlda.n.SQLTYPE		2	3		5				
	sqlda.n.SQLLEN		2	3		5				
	sqlda.n.SQLDATA	1	2	3			6		8	
	sqlda.n.SQLIND	1	2	3			6		8	9
	sqlda.n.SQLNAME.length						6	7		
	sqlda.n.SQLNAME.data						6	7		
	sqlda.n.SQLDATATYPE_NAME		2	3		5				
	sqlda.n.SQLLONGLEN		2	3		5				
	sqlda.n.SQLDATALEN	1	2	3			6	7		
SQLCA (all elements)							6	7		

Note:

Before invoking the stored procedure, the client application must:

1. Allocate storage for the pointer element based on SQLTYPE and SQLLEN.
2. Initialize the element with the appropriate data.

When called by the application, the database manager:

3. Sends data in the original element to a duplicate element allocated at the stored procedure. The SQLN element is initialized with the data in the SQLD element.

When invoked, the stored procedure can:

4. Alter data in the duplicate element. The data can be altered as needed since it is not checked for validity or returned to the client application.

When the stored procedure terminates, the database manager:

5. Checks data in the duplicate elements. If the values in these fields do not match the data in the original elements, an error is returned.
6. Returns data in the duplicate elements to the original element.
7. The data can be altered as needed since it is not checked for validity.
8. The data pointed to by the elements can be altered as needed since they are not checked for validity but are returned to the client application.
9. The SQLIND field is not passed in or out if SQLTYPE indicates the column type is not nullable.

Building the Stored Procedure Application

Stored procedure applications have special compile and link requirements. The client procedure must be part of an executable file, while the stored procedure must be placed in a library on the database server.

When testing these applications, it is helpful to execute both procedures locally on the same machine. (This machine must be configured as a client/server.) After the application executes locally without error, move the stored procedure to a database server, or move the client procedure to a database client for further testing.

Client Application

The precompile, compile, and link requirements of the client application are identical to those of a normal database manager application.

Refer to the *DB2 SDK Building Applications* book for your environment for details on how to build and execute a client application in the programming language you wish to use.

Stored Procedure

The stored procedure must be precompiled, compiled, and linked to produce a library. Special compile options are required, and the application must be bound to the database on the server before it can be run. Remember also that for all these steps, the procedure must use the same code page as the database.

Refer to the *DB2 SDK Building Applications* book for your server environment for details on how to prepare your server procedure.

Resolving Problems

If a stored procedure application fails to execute properly, ensure that:

- The stored procedure is built using the correct calling sequence, compile options, and so on.
- The application executes locally with both client application and stored procedure on the same workstation.
- The stored procedure is installed in the proper location in accordance with the instructions in the *DB2 SDK Building Applications* book for your operating system.

  For example, in an OS/2 environment, the dynamic link library containing the stored procedure is located in a directory identified in the LIBPATH statement in the CONFIG.SYS file of the database server, or the calling function provides a full path. In an AIX- or UNIX-based environment, the library containing the stored procedure is located in the `$HOME/sqllib/function` directory of the DB2 instance owner, or the calling function provides a full path.
- The application is bound to the database.
- The stored procedure accurately returns any SQLCA error information to the client application.
- The stored procedure should not allocate its own SQLDA or sqlvar storage. The stored procedure should use the storage passed into the procedure by parameters.
- Stored procedure function names are *case-sensitive*, and must match exactly on client and server.

You can use the debugger supplied with your compiler to debug a stored procedure as you would any other application. Consult your compiler documentation for information on using the supplied debugger.

As an example, the following are the high level steps for using the IPMD debugger supplied with IBM VisualAge C++ on OS/2:

- Compile the source file for the stored procedure DLL with the `-Ti+` and `-o-` flags, and then link the DLL using the `-DEBug` option.
- Copy the resulting DLL to the `%DB2PATH%\DLL` directory (or any other directory specified by the LIBPATH environment variable).
- The stored procedure is run as a not-fenced stored procedure.
- Debug the stored procedure on the database server by invoking the client application on the same machine. Assuming the client is `OUTCLI` and the server is `OUTSRV`, the steps are as follows:

  ```
  ipmd outcli.exe
  ```

- After the debugger starts up, click on BREAKPOINTS and then LOAD OCCURRENCE.
- Enter the name of the DLL in LOAD OCCURRENCE; for example, OUTSRV.
- Click on RUN. The debugger will stop when the DLL is loaded.
- At this point, you can set the break point in the stored procedure (OUTSRV.DLL).

Refer to the VisualAge C++ product documentation for further information on using the IPMD debugger.

Working with Not-Fenced Stored Procedures

Your stored procedure can run as a *not-fenced* or *fenced* stored procedure. A *not-fenced* stored procedure runs in the same address space as the database manager (the DB2 Agent's address space). A fenced stored procedure called from the server, runs in an address space (the application's address space) that is isolated from the database manager's address space. A fenced stored procedure called from a remote machine runs in a special DB2 process whose address space is distinct from the DB2 System Controller. Running your stored procedure as not-fenced results in increased performance when compared with running it as fenced.

This performance increase is realized regardless of whether the client application runs locally or remotely from the server machine where you are running your stored procedure application. Note that while performance improvements can be expected when running a not-fenced stored procedure, there is the possibility that user code could accidentally or maliciously damage the database control structures. In addition, local fenced stored procedures are easier for you to debug as the stored procedures run in the application's address space. Thus, when debugging your application, the debugger will have access to the stored procedure code. With not-fenced stored procedures, the debugger will also have access to the database manager's address space, thus complicating your debugging activity.

Note: Due to the associated risk of damaging your database, you should only use not-fenced stored procedures when you need to maximize the performance benefits. In addition, ensure that all your stored procedures are thoroughly tested prior to running them as not-fenced. If a severe error does occur while you are running a not-fenced stored procedure, the database manager determines whether the error occurred in the stored procedure code or the database code, and attempts an appropriate recovery.

Keep in mind when you are writing a not-fenced stored procedure, that it may run in a threaded environment, depending on the operating system. Thus, the stored procedure must either be completely re-entrant, or manage its static variables so that access to these variables is serialized.

Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. See “The WCHARTYPE Precompiler Option” on page 441 for more information.

You can indicate that a stored procedure is fenced or not-fenced by placing it in a special directory. See the *DB2 SDK Building Applications* book for your operating system for information on where to place your not-fenced stored procedure.

The following not-fenced stored procedures are not supported by DB2:

- 16-bit
- REXX
- Multi-threaded.

The following DB2 APIs and any DB2 CLI API are not supported in a not-fenced stored procedure:

- BIND
- EXPORT
- IMPORT
- PRECOMPILE PROGRAM
- ROLLFORWARD DATABASE.

Example Output-SQLDA Programs

Following is a sample program demonstrating the use of an output SQLDA structure. The client application invokes a stored procedure that determines the median salary for employees in the SAMPLE database. (The definition of the median is that half the values lie above it, and half below it.) The median salary is then passed back to the client application using an output SQLDA structure.

This sample program calculates the median salary of all employees in the SAMPLE database. Since there is no existing SQL column function to calculate medians, the median salary can be found iteratively by the following algorithm:

1. Determine the number of records, *n*, in the table.
2. Order the records based upon salary.
3. Fetch records until the record in row position *n/2+1* is found.
4. Read the median salary from this record.

An application that uses neither the stored procedures technique, nor blocking cursors must FETCH each salary across the network as shown in Figure 20.

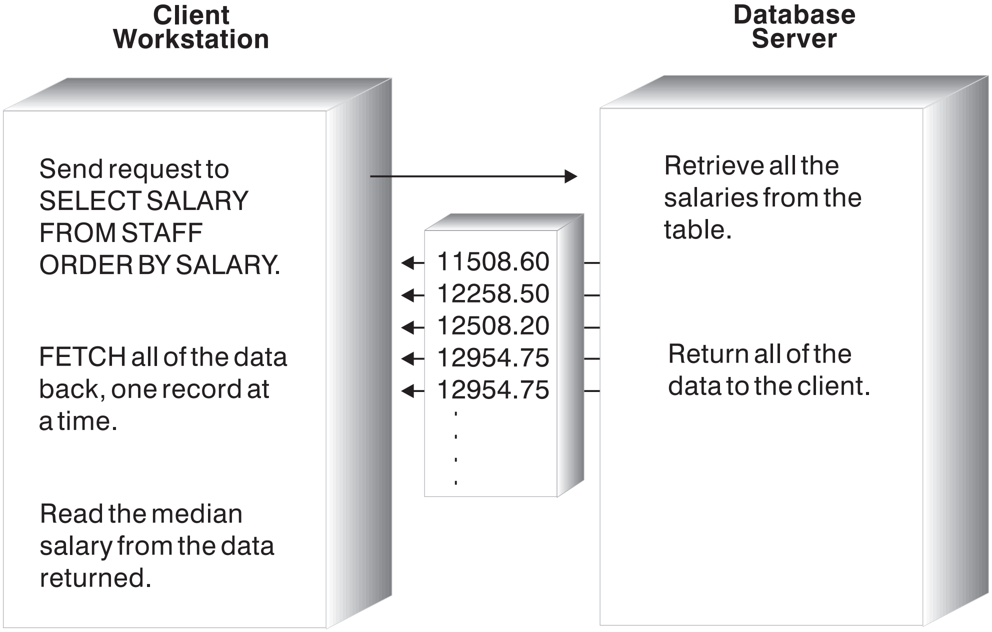

Client Workstation
Database Server
Send request to SELECT SALARY FROM STAFF ORDER BY SALARY.
FETCH all of the data back, one record at a time.
Read the median salary from the data returned.
11508.60
12258.50
12508.20
12954.75
12954.75
Retrieve all the salaries from the table.
Return all of the data to the client.

Figure 20. Median Sample Without a Stored Procedure

Since only the salary at row *n/2+1* is needed, the application discards all the additional data, *but only after it is transmitted across the network*.

An application using the stored procedures technique can be designed to allow the stored procedure to process and discard the unnecessary data, returning only the median salary to the client application. Figure 21 shows this feature.

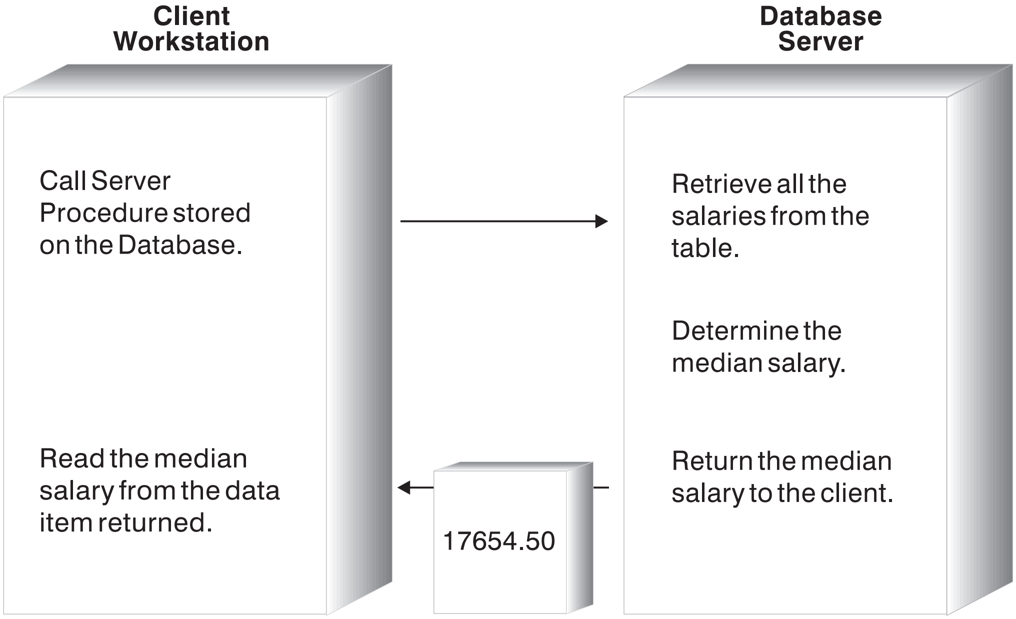

Client Workstation
Database Server
Call Server Procedure stored on the Database.
Retrieve all the salaries from the table.
Determine the median salary.
Return the median salary to the client.
Read the median salary from the data item returned.
17654.50

Figure 21. Median Sample Using a Stored Procedure

“How the Example Output-SQLDA Client Application Works” on page 183 shows a sample output-SQLDA client application and sample output-SQLDA stored procedure in some of the supported languages.

How the Example Output-SQLDA Client Application Works

1. **Include Files**. The program begins with the following include files:

`SQL`	Defines the symbol `SQL_TYP_FLOAT`
`SQLDA`	Defines the descriptor area
`SQLCA`	Defines the communication area for error handling

2. **Allocate the output SQLDA.** This step declares and allocates an SQLDA structure. One SQLVAR element is used to return data from the server procedure.
3. **Declare local variables**. The SQLVAR element of the SQLDA is initialized to point to these variables. These variables contain the data returned by the server procedure.
4. **Initialize the Output SQLDA**. The following fields of the output SQLDA are initialized:
   - The SQLN and SQLD elements are set to the total number of SQLVAR elements allocated and used.
   - The SQLTYPE and SQLLEN elements are set to indicate FLOAT data type for C and FORTRAN, and DECIMAL for COBOL.
   - The SQLDATA and SQLIND elements are set to point at the local variables `sal` and `salind` declared previously. When data is placed in these fields, it can be referenced by `sal` and `salind`.
5. **Connect to Database**. The application executes the CONNECT TO statement requesting shared access. This must be done before invoking the server procedure.
6. **Invoke the Server Procedure**. The application invokes the procedure `outsrv` at the location of the database, `sample`, using:
   a. CALL statement with host variables
   b. CALL statement with an SQLDA.
7. **Disconnect from Database**.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C `check_error` is redefined as `CHECKERR` and is located in the `util.c` file.

COBOL `CHECKERR` is an external program named `checkerr.cbl`.

FORTRAN `CHECKERR` is a subroutine located in the `util.f` file.

REXX `CHECKERR` is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Example: OUTCLI.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h> 1
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include "util.h"

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
     char database[9];
     char userid[9];
     char passwd[19];
     /* Declare a Local Variable for Holding the Procedure's Name */
     char procname[255] = "outsrv";

     /* Declare Local Variables for Holding Returned Data */
     double sal     = 0.0;  3
     short  salind  = 0;
   EXEC SQL END DECLARE SECTION;


   /* Declare the output SQLDA */
   struct sqlda *inout_sqlda = (struct sqlda *)
   malloc(SQLDASIZE(1)); 2

   /* Declare the SQLCA */
   struct sqlca sqlca;

   if (argc != 4) {
      printf ("\nUSAGE: outcli remote_database userid passwd\n\n");
      return 1;
   }
   strcpy (database, argv[1]);
   strcpy (userid, argv[2]);
   strcpy (passwd, argv[3]);
   /* Connect to Remote Database */
   printf("CONNECT TO Remote Database.\n");
   EXEC SQL CONNECT TO :database USER :userid USING :passwd; 5
   CHECKERR ("CONNECT TO RSAMPLE");

   /*****************************************************\
   * Call the Remote Procedure via CALL with Host Variables *
   \*****************************************************/
   printf("Use CALL with Host Variable to invoke the Server Procedure "
      "named outsrv\n");
   salind = -1;                  /* Sal has no input, so set to null */
   EXEC SQL CALL :procname (:sal :salind);  6a
   CHECKERR ("CALL WITH HOST VARIABLES");
   printf("Server Procedure Complete.\n");

   /* Print Salary Returned in The Host Variables */
   printf("Median Salary = %.2f\n\n", sal );
   /***********************************************\
   * Call the Remote Procedure via CALL with SQLDA *
   \***********************************************/
   /* Initialize the output SQLDA */   4
   inout_sqlda->sqln = 1;
   inout_sqlda->sqld = 1;
   inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;
   inout_sqlda->sqlvar[0].sqllen  = sizeof( double );
   inout_sqlda->sqlvar[0].sqldata = (char *)&sal;
   inout_sqlda->sqlvar[0].sqlind  = (short *)&salind;

   printf("Use CALL with SQLDA to invoke the Server Procedure "
      "named outsrv\n");
   salind = -1;                  /* Sal has no input, so set to null */
   EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;   6b
   CHECKERR ("CALL WITH SQLDA");
   printf("Server Procedure Complete.\n");

   /* Print Salary Returned in The Host Variables */
   printf("Median Salary = %.2f\n\n", sal );

   /* Free allocated memory */
   free( inout_sqlda );

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;  7
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : outcli.sqc */
```

COBOL Example: OUTCLI.SQB

```
 Identification Division.
 Program-ID. "outcli".

 Data Division.
 Working-Storage Section.

* Copy Files for Constants and Structures.
 copy "sql.cbl". 1
 copy "sqlenv.cbl".
 copy "sqlca.cbl".

 01 decimal-sqllen     pic s9(4) comp-5.
 01 decimal-parts      redefines decimal-sqllen.
     05 precision      pic x.
     05 scale          pic x.

* Declare an Output SQLDA Structure.
 01  io-sqlda sync. 2
     05 io-sqldaid     pic x(8) value "O-DA   ".
     05 io-sqldabc     pic s9(9) comp-5.
     05 io-sqln        pic s9(4) comp-5.
     05 io-sqld        pic s9(4) comp-5.
     05 io-sqlvar occurs 1 to 99 times
        depending on io-sqld.
        10 io-sqltype  pic s9(4) comp-5.
        10 io-sqllen   pic s9(4) comp-5.
        10 io-sqldata  usage is pointer.
        10 io-sqlind   usage is pointer.
        10 io-sqlname.
           15 io-sqlnamel   pic s9(4) comp-5.
           15 io-sqlnamec   pic x(30).

 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 dbname            pic x(8). 3
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).

* Declare Variables for the SQL CALL.
 77  prog-name       pic x(19)        value "outsrv".

* Declare Local Variables for Holding Actual Data.
 77  salary          pic s9(5)v99  comp-3.
 77  sal-ind         pic s9(4)     comp-5.

 EXEC SQL END DECLARE SECTION END-EXEC.

* Declare Output Mask for Salary
 77  sal-out         pic z9(5).99-.

* Declare a Null Pointer Variable.
 77  null-ptr-int    pic s9(9)     comp-5.
 77  null-ptr redefines null-ptr-int pointer.

 77 errloc pic x(80).
 Procedure Division.
 Main Section.

* Initialize the Input/Output SQLDA Structure
     move 1 to io-sqln.  4
     move 1 to io-sqld.
     move sql-typ-ndecimal to io-sqltype(1).
* Length = 7 digits precision and 2 digits scale
     move x"07" to precision.
     move x"02" to scale.
     move decimal-sqllen to io-sqllen(1).
     set io-sqldata(1) to address of salary.
     set io-sqlind(1)  to address of sal-ind.

* CONNECT TO DATABASE
     display "Enter in the database name : " with no advancing.
     accept dbname.

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO :dbname USER :userid USING :pa 5
        END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Call the Remote Procedure.
     display "Use CALL with Host Var. to invoke Server Procedure."
     EXEC SQL CALL :prog-name (:SALARY:SAL-IND) END-EXE 6a
     move "CALL HV" to errloc.
     call "checkerr" using SQLCA errloc.
     display "Server Procedure Complete.".

* Print Salary Returned in the host variable.
     move salary to sal-out.
     display "Median Salary = " sal-out.
     display " ".

* Call the Remote Procedure.
     display "Use CALL with SQLDA to invoke Server Procedure."
     EXEC SQL CALL :prog-name USING DESCRIPTOR  6b
                              :IO-SQLDA END-EXEC.
     move "CALL DA" to errloc.
     call "checkerr" using SQLCA errloc.
     DISPLAY "Server Procedure Complete.".

* Print Salary Returned in IO-SQLDA.
```

```
     move salary to sal-out.
     display "Median Salary = " sal-out.

* Disconnect from Remote Database.
     EXEC SQL CONNECT RESET END-EXEC.  7
     stop run.
     exit.
```

FORTRAN UNIX Example: OUTCLI.SQF

```
      program outcli
      implicit none

*     Copy Files for Constants and Structures
      include 'sql.f'
      include 'sqlenv.f'  1
      include 'sqlutil.f'
      include 'sqldact.f'

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION  3
        character*8    dbname
        character*8    userid
        character*18   passwd
        real*8         salary
        integer*2      sal_ind
        character*19   prog_name
      EXEC SQL END DECLARE SECTION

      integer*4        rc
      character*80     errloc

      integer*2        sqlvar1
      parameter        (sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz)

      character        io_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)  2
      character*8      io_sqldaid
      integer*4        io_sqldabc
      integer*2        io_sqln
      integer*2        io_sqld

      integer*2        io_sqltype1    ! First Variable
      integer*2        io_sqllen1
      integer*4        io_sqldata1
      integer*4        io_sqlind1
      integer*2        io_sqlnamelen1
      character*30     io_sqlnamedata1

      equivalence (io_sqlda(sqlda_sqldaid_ofs), io_sqldaid)
      equivalence (io_sqlda(sqlda_sqldabc_ofs), io_sqldabc)
      equivalence (io_sqlda(sqlda_sqln_ofs), io_sqln)
      equivalence (io_sqlda(sqlda_sqld_ofs), io_sqld)

      equivalence (io_sqlda(sqlvar1+sqlvar_type_ofs), io_sqltype1)
      equivalence (io_sqlda(sqlvar1+sqlvar_len_ofs), io_sqllen1)
      equivalence (io_sqlda(sqlvar1+sqlvar_data_ofs), io_sqldata1)
      equivalence (io_sqlda(sqlvar1+sqlvar_ind_ofs), io_sqlind1)
      equivalence (io_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             io_sqlnamelen1)
      equivalence (io_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             io_sqlnamedata1)

      io_sqldaid = 'O_SQLDA'  4
      io_sqldabc = sqlda_header_sz + 1*sqlvar_struct_sz
      io_sqln    = 1
      io_sqld    = 1
      io_sqltype1 = sql_typ_nfloat
      io_sqllen1  = 8
      rc = sqlgaddr (%ref(salary), %ref(io_sqldata1))
      rc = sqlgaddr (%ref(sal_ind), %ref(io_sqlind1))


*     Program Logic

      print *, 'Enter in the remote database to attach to:'
      read 100, dbname
100   format (a8)

      print *, 'Enter your user id (default none):'
      read 100, userid

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO :dbname
      else
      print *, 'Enter your password :'
      read 100, passwd

        print *, 'CONNECT TO Remote Database.'
      EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
      end if  5
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

      prog_name = 'outsrv'

      print *, ' '
      print *, 'Use CALL with Host Variable to invoke the Server Procedu
     cre named outsrv.'
      EXEC SQL CALL :prog_name (:salary:sal_ind)  6
      errloc = 'CALL HV'
      call checkerr (sqlca, errloc, *999)
      print *, 'Server Procedure Complete.'

*     Print Salary Returned in OUTPUT SQLDA
      print *, 'Median Salary = ', salary

      print *, ' '
      print *, 'Use CALL with SQLDA to invoke the Server Procedure named
     c outsrv.'
      EXEC SQL CALL :prog_name USING DESCRIPTOR :io_sqlda  6
      errloc = 'CALL DA'
      call checkerr (sqlca, errloc, *999)
      print *, 'Server Procedure Complete.'

*     Print Salary Returned in OUTPUT SQLDA
      print *, 'Median Salary = ', salary

*     Disconnect from Remote Database.
      EXEC SQL CONNECT RESET  7
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

999   stop
      end
```

REXX Example: OUTCLI.CMD

```
/* REXX OUTput CLIent  */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if rxfuncquery('SQLDBS') <> 0 then
     rcy = rxfuncadd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if rxfuncquery('SQLEXEC') <> 0 then
     rcy = rxfuncadd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")

/* pull in command line arguments */
parse arg dbname userid passwd .

/* check to see if the proper number of arguments have been passed in */
if passwd = '' then
do
  say 'USAGE : outcli dbname userid passwd'
  exit
end

procname = 'outsrv.cmd'
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */   4
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485           /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale  = 2       /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision  = 7   /* WIDTH OF DECIMAL  */
io_sqlda.1.sqldata = 00000.00      /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1             /* NO INPUT DATA */

/* CONNECT TO REMOTE DATABASE */
say 'Connect to Remote Database.'
call SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' passwd  5
call CHECKERR 'CONNECT'

/* CALL THE REMOTE PROCEDURE USING CALL AND HOST VARIABLES */
say "Use CALL with Host Variables to invoke the Stored Procedure outsrv"
sal=00000.00 /* To force precision 7, scale 2 */
salind=0
call SQLEXEC 'CALL :procname (:sal :salind)'   6a
call CHECKERR 'CALL USING HV'
say 'Median Salary = ' sal

/* CALL THE REMOTE PROCEDURE USING CALL AND SQLDA */
say "Use the CALL with SQLDA to invoke the Stored Procedure outsrv"
call SQLEXEC 'CALL :procname USING DESCRIPTOR :io_sqlda'   6b
call CHECKERR 'CALL USING SQLDA'
say 'Median Salary = ' io_sqlda.1.sqldata

/* DISCONNECT FROM REMOTE DATABASE */
call SQLEXEC 'CONNECT RESET'   7
call CHECKERR 'CONNECT RESET'
```

```
exit 0


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

How the Example Output-SQLDA Stored Procedure Works

1. **Include Files**. The SQL include file defines the symbol SQLZ_DISCONNECT_PROC.
2. **Declare Server Procedure**. The server procedure, outsrv, is declared.

   The procedure accepts pointers to SQLDA and SQLCA structures.
3. **Determine Median Position**. The row at position cnt/2+1 of the cursor declared previously is assumed to contain the median value.
4. **Fetch Median**. The server procedure issues FETCH statements until the row containing the median salary is reached. The FETCH statement uses the SQLDA provided by the client application.
5. **Return to the Client Application**. Copy the SQLCA to the SQLCA of the client application and return the value SQLZ_DISCONNECT_PROC, indicating that no further calls to the server procedure will be made.

   If an error occurs, the SQLCA containing the error information is copied to the SQLCA of the client application and a ROLLBACK statement is issued.

Note: Server procedures cannot be written in REXX on AIX systems.

C Example: OUTSRV.SQC

```
#include <memory.h>
#include <string.h>
#include <sqlenv.h>     1
#include <sql.h>
#include <sqlda.h>

SQL_API_RC SQL_API_FN outsrv (    2
    void *reserved1,
    void *reserved2,
    struct sqlda   *inout_sqlda,
    struct sqlca   *ca) {

   /* Declare a local SQLCA */
   EXEC SQL INCLUDE SQLCA;

   /* Declare Host Variables */
   EXEC SQL BEGIN DECLARE SECTION;
      short num_records;
      char  stmt[512];
   EXEC SQL END DECLARE SECTION;

   /* Declare Miscellaneous Variables */
   int counter = 0;
   EXEC SQL WHENEVER SQLERROR   GOTO error_exit;
   EXEC SQL WHENEVER SQLWARNING CONTINUE;
   EXEC SQL DECLARE c1 CURSOR FOR s1;

   /* Prepare a Statement to Obtain and Order all Salaries */
   strcpy( stmt, "SELECT salary FROM STAFF ORDER BY salary" );
   EXEC SQL PREPARE s1 FROM :stmt;
   /*Determine the Total Number of Records */
   EXEC SQL SELECT COUNT(*) INTO :num_records FROM STAFF;  3

   /* Fetch Salaries until the Median Salary is Obtained */
   EXEC SQL OPEN c1;
   while ( counter++ < num_records/2 + 1 )   4
      EXEC SQL FETCH c1 USING DESCRIPTOR :*inout_sqlda;
   EXEC SQL CLOSE c1;
   EXEC SQL COMMIT;

   /* Return the SQLCA to the Calling Program */  5
   memcpy( ca, &sqlca, sizeof( struct sqlca ) );
   return(SQLZ_DISCONNECT_PROC);

error_exit:
   /* An Error has occurred -- ROLLBACK and return to Calling Program */
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   memcpy( ca, &sqlca, sizeof( struct sqlca ) );
   EXEC SQL ROLLBACK;

   return(SQLZ_DISCONNECT_PROC);
}
```

COBOL Example: OUTSRV.SQB

```
 Identification Division.
 Program-ID. "outsrv".

 Data Division.
 Working-Storage Section.

* Copy Files for Constants and Structures
 copy "sql.cbl".
 copy "sqlenv.cbl".  1
 copy "sqlca.cbl".

* Declare Host Variables
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   77  num-records  pic s9(9) comp-5 value 0.
   01  stmt.
       49 stmt-len pic s9(4) comp-5.
       49 stmt-str pic x(50).
 EXEC SQL END DECLARE SECTION END-EXEC.

* Declare Miscellaneous Variables
 77  cntr1 pic s9(9) comp-5 value 0.
 77  cntr2 pic s9(9) comp-5 value 0.

 Linkage Section.

* Declare Parameters
 77  reserved1   pointer.
 77  reserved2   pointer.
 77  inout-sqlda pointer.

 01 O-SQLCA SYNC.
    05 SQLCAID PIC X(8).
    05 SQLCABC PIC S9(9) COMP-5.
    05 SQLCODE PIC S9(9) COMP-5.
    05 SQLERRM.
       49 SQLERRML PIC S9(4) COMP-5.
       49 SQLERRMC PIC X(70).
    05 SQLERRP PIC X(8).
    05 SQLERRD OCCURS 6 TIMES PIC S9(9) COMP-5.
    05 SQLWARN.
       10 SQLWARN0 PIC X.
       10 SQLWARN1 PIC X.
       10 SQLWARN2 PIC X.
       10 SQLWARN3 PIC X.
       10 SQLWARN4 PIC X.
       10 SQLWARN5 PIC X.
       10 SQLWARN6 PIC X.
       10 SQLWARN7 PIC X.
       10 SQLWARN8 PIC X.
       10 SQLWARN9 PIC X.
       10 SQLWARNA PIC X.
    05 SQLSTATE PIC X(5).

 Procedure Division using reserved1 reserved2 inout-sqlda O-SQLCA. 2

 Main Section.
     EXEC SQL WHENEVER SQLERROR GOTO ERROR-EXIT END-EXEC.
     EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.

     EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.

* Prepare a Statement to Obtain and Order all Salaries.
     move 50 to stmt-len.
     move "SELECT salary FROM staff ORDER BY salary" to stmt-str.
     EXEC SQL PREPARE S1 FROM :stmt END-EXEC.

* Determine Total Number of Records.
     EXEC SQL SELECT COUNT(*) INTO :num-records  3
              FROM staff END-EXEC.

     EXEC SQL OPEN C1 END-EXEC.

* Fetch Salaries until the Median Salary is Obtained.
     compute cntr2 = 1 + num-records / 2.
     perform Fetch-Row until cntr1 = cntr2. 4

     EXEC SQL CLOSE C1 END-EXEC.

     EXEC SQL COMMIT END-EXEC.

* Return the SQLCA Information to the Calling Program.
     move SQLCA to O-SQLCA.
     EXEC SQL COMMIT END-EXEC.
     move SQLZ-DISCONNECT-PROC to return-code
     goback.  5

 Fetch-Row.
     add 1 to cntr1.
     EXEC SQL FETCH C1 USING DESCRIPTOR :inout-sqlda END-EXEC.

 ERROR-EXIT.
* An Error has Occurred -- ROLLBACK and Return to Calling Program.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     move SQLCA to O-SQLCA.
     EXEC SQL ROLLBACK END-EXEC.
     move SQLZ-DISCONNECT-PROC to return-code
     goback.
```

FORTRAN UNIX Example: OUTSRV.SQF

```
      integer*4 function outsrv( reserved1, 2
     +                           reserved2,
     +                           io_sqlda,
     +                           ca )
      implicit none

*     Include Files for Constants and Structures
      include 'sql.f' 1
      include 'sqlenv.f'
      include 'sqlutil.f'
      include 'sqldact.f'

*     Declare Input Parameters
      integer*4  reserved1
      integer*4  reserved2
      character  io_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)
      character  ca(sqlca_size)

*     Declare a Local SQLCA
      EXEC SQL INCLUDE SQLCA

*     Declare Host Variables
      EXEC SQL BEGIN DECLARE SECTION
        integer*4      num_records
        character*50   stmt
      EXEC SQL END DECLARE SECTION

*     Declare Miscellaneous Variables
      integer*2    cntr

*     Program Logic

      EXEC SQL WHENEVER SQLERROR GOTO 100

      EXEC SQL DECLARE c1 CURSOR FOR s1

*     Prepare a Statement to Obtain and Order all Salaries
      stmt = 'SELECT SALARY FROM STAFF ORDER BY SALARY'
      EXEC SQL PREPARE s1 FROM :stmt

*     Determine the Total Number of Records
      EXEC SQL SELECT COUNT(*) INTO :num_records FROM STAFF 3

*     Fetch Salaries until the Median Salary is Obtained
      EXEC SQL OPEN c1
      cntr = 0
      do 10 while ( cntr .lt. (num_records/2 + 1) )
        cntr = cntr + 1
        EXEC SQL FETCH c1 USING DESCRIPTOR :io_sqlda 4
10    end do
      EXEC SQL CLOSE c1

      EXEC SQL COMMIT

*     Copy Local Variables into Dummy Output Variables
*     Copy the sqlca to the ca
      cntr = 0
      do 20 while ( cntr .lt. sqlca_size )
        cntr = cntr + 1
        ca(cntr) = sqlca(cntr)
20    end do

      goto 200

100   continue                                ! An Error has Occurred
*     Copy Local Variables into Dummy Output Variables
*     Copy the sqlca to the ca
      EXEC SQL WHENEVER SQLERROR CONTINUE
      cntr = 0
      do 30 while ( cntr .lt. sqlca_size )
        cntr = cntr + 1
        ca(cntr) = sqlca(cntr)
30    end do

      EXEC SQL ROLLBACK

200   continue                                 ! Exit Program
      outsrv = sqlz_disconnect_proc
      return 5
      end
```

REXX OS/2 Example: OUTSRV.CMD

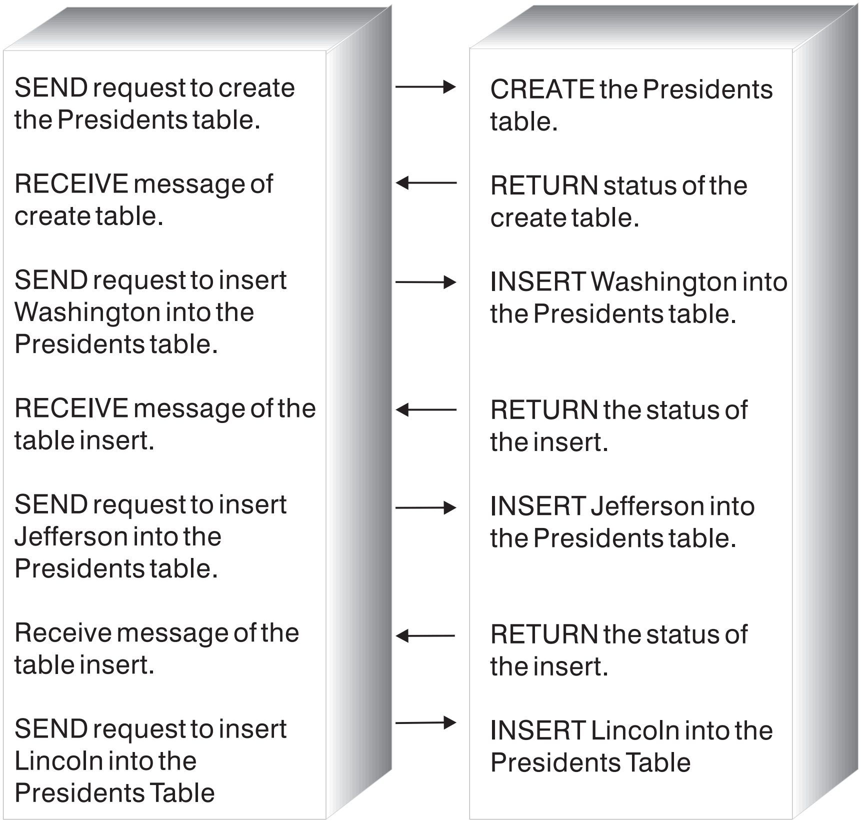
```
/* REXX OUTput SeRVer */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if rxfuncquery('SQLDBS') <> 0 then
     rcy = rxfuncadd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if rxfuncquery('SQLEXEC') <> 0 then
     rcy = rxfuncadd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'
call CHECKERR 'DECLARE'

/* DETERMINE TOTAL NUMBER OF RECORDS */
stmt = 'SELECT COUNT(*) FROM STAFF'   3
call SQLEXEC 'PREPARE s1 FROM :stmt'
call CHECKERR 'PREPARE count statement'

call SQLEXEC 'OPEN c1'
call CHECKERR 'OPEN count statement'

call SQLEXEC 'FETCH c1 INTO :num_records'
call CHECKERR 'FETCH'

call SQLEXEC 'CLOSE c1'
call CHECKERR 'CLOSE c1 after FETCH of num_records'

/* PREPARE A STATEMENT TO OBTAIN AND ORDER ALL SALARIES */
stmt = 'SELECT salary FROM STAFF ORDER BY salary'
call SQLEXEC 'PREPARE s1 FROM :stmt'
call CHECKERR 'PREPARE order statement'

call SQLEXEC 'OPEN c1'
call CHECKERR 'OPEN order statement'

/* FETCH SALARIES UNTIL THE MEDIAN SALARY IS OBTAINED */
i = 0
do while ( i < (num_records/2) )   4
  call SQLEXEC 'FETCH c1 USING DESCRIPTOR :SQLRODA'
  call CHECKERR 'FETCH'
  i = i + 1
end
call SQLEXEC 'CLOSE c1'
call CHECKERR 'CLOSE c1 after FETCH of median'

call SQLEXEC 'COMMIT'
call CHECKERR 'COMMIT'

exit 0 /* Normal exit */


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Example Input-SQLDA Programs

Following is a sample program demonstrating the use of an input SQLDA structure. The client application invokes a stored procedure that creates a table named `Presidents` and loads the table with data.

This program creates a table called `Presidents` in the SAMPLE database. It then inserts the values `Washington`, `Jefferson`, and `Lincoln` into the table.

Without using stored procedures, the sample program would have been designed to transmit data across the network in four separate requests in order to process each SQL statement, as shown in Figure 22.



SEND request to create the Presidents table. → CREATE the Presidents table.

RECEIVE message of create table. ← RETURN status of the create table.

SEND request to insert Washington into the Presidents table. → INSERT Washington into the Presidents table.

RECEIVE message of the table insert. ← RETURN the status of the insert.

SEND request to insert Jefferson into the Presidents table. → INSERT Jefferson into the Presidents table.

Receive message of the table insert. ← RETURN the status of the insert.

SEND request to insert Lincoln into the Presidents Table → INSERT Lincoln into the Presidents Table

Figure 22. Input-SQLDA Sample Without a Stored Procedure

Instead, the sample program makes use of the stored procedures technique to transmit all of the data across the network in one request, allowing the server procedure to execute the SQL statements as a group. This technique is shown in Figure 23.

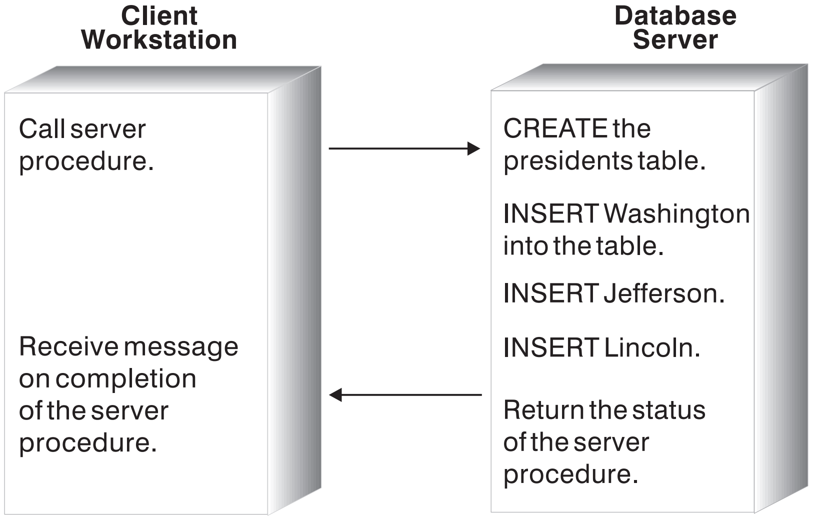

Client Workstation
Database Server
Call server procedure.
Receive message on completion of the server procedure.
CREATE the presidents table.
INSERT Washington into the table.
INSERT Jefferson.
INSERT Lincoln.
Return the status of the server procedure.

Figure 23. Input-SQLDA Sample With a Stored Procedure

A sample input-SQLDA client application and sample input-SQLDA stored procedure is shown on 203.

How the Example Input-SQLDA Client Application Works

1. **Initialize the Input SQLDA Structure**. The following fields of the input SQLDA are initialized:

   - The SQLN and SQLD elements are set to the total number of SQLVAR elements allocated and used.
   - The SQLTYPE elements are set to indicate character data type.
   - The first SQLDATA element is set to the name of the table. The second through fourth SQLDATA elements are set to the values `Washington`, `Jefferson`, and `Lincoln`.
   - The SQLLEN elements are set to the length of each SQLDATA element (plus 1 byte for the C language null terminator).
   - The SQLIND elements are set to `NULL`.

2. **Invoke the Server Procedure**. The application invokes the procedure `inpsrv` at the location of the database, `sample` using:

   a. CALL statement with host variables
   b. CALL statement with an SQLDA.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C `check_error` is redefined as `CHECKERR` and is located in the `util.c` file.

COBOL `CHECKERR` is an external program named `checkerr.cbl`.

FORTRAN `CHECKERR` is a subroutine located in the `util.f` file.

REXX `CHECKERR` is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Example: INPCLI.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqlutil.h>
#include "util.h"

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
     char database[9];
     char userid[9];
     char passwd[19];
     char procname[255]  = "inpsrv";
     char table_name[11]    = "PRESIDENTS";
     char data_item0[21]    = "Washington";
     char data_item1[21]    = "Jefferson";
     char data_item2[21]    = "Lincoln";
     short tableind, dataind0, dataind1, dataind2;
   EXEC SQL END DECLARE SECTION;

   /* Declare Variables for CALL USING */
   struct sqlca    sqlca;
   struct sqlda   *inout_sqlda = NULL;

   if (argc != 4) {
      printf ("\nUSAGE: inpcli remote_database userid passwd\n\n");
      return 1;
   }

   strcpy (database, argv[1]);
   strcpy (userid, argv[2]);
   strcpy (passwd, argv[3]);
   /* Connect to Remote Database */
   printf("CONNECT TO Remote Database.\n");
   EXEC SQL CONNECT TO :database USER :userid USING :passwd;
   CHECKERR ("CONNECT TO SAMPLE");

   /********************************************************\
   * Call the Remote Procedure via CALL with Host Variables *
   \********************************************************/
   printf("Use CALL with Host Variable to invoke the Server Procedure"
      " named inpsrv.\n");
   tableind = dataind0 = dataind1 = dataind2 = 0;

   EXEC SQL CALL :procname (:table_name:tableind, :data_item0:dataind0,
      :data_item1:dataind1, :data_item2:dataind2);  2a
   CHECKERR ("CALL WITH HOST VARIABLE");
   printf("Server Procedure Complete.\n\n");


   /* Allocate and Initialize Input SQLDA */    1
   inout_sqlda = (struct sqlda *)malloc( SQLDASIZE(4) );
   inout_sqlda->sqln = 4;
   inout_sqlda->sqld = 4;

   inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NCSTR;
   inout_sqlda->sqlvar[0].sqldata = table_name;
   inout_sqlda->sqlvar[0].sqllen  = strlen( table_name ) + 1;
   inout_sqlda->sqlvar[0].sqlind  = &tableind;

   inout_sqlda->sqlvar[1].sqltype = SQL_TYP_NCSTR;
   inout_sqlda->sqlvar[1].sqldata = data_item0;
   inout_sqlda->sqlvar[1].sqllen  = strlen( data_item0 ) + 1;
   inout_sqlda->sqlvar[1].sqlind  = &dataind0;

   inout_sqlda->sqlvar[2].sqltype = SQL_TYP_NCSTR;
   inout_sqlda->sqlvar[2].sqldata = data_item1;
   inout_sqlda->sqlvar[2].sqllen  = strlen( data_item1 ) + 1;
   inout_sqlda->sqlvar[2].sqlind  = &dataind0;

   inout_sqlda->sqlvar[3].sqltype = SQL_TYP_NCSTR;
   inout_sqlda->sqlvar[3].sqldata = data_item2;
   inout_sqlda->sqlvar[3].sqllen  = strlen( data_item2 ) + 1;
   inout_sqlda->sqlvar[3].sqlind  = &dataind0;

   /***********************************************\
   * Call the Remote Procedure via CALL with SQLDA *
   \***********************************************/
   printf("Use CALL with SQLDA to invoke the Server Procedure named "
      "inpsrv.\n");
   tableind = dataind0 = dataind1 = dataind2 = 0;

   EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;   2b
   CHECKERR ("CALL WITH SQLDA");
   printf("Server Procedure Complete.\n\n");

   /* Free allocated memory */
   free( inout_sqlda );

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : inpcli.sqc */
```

COBOL Example: INPCLI.SQB

```
 Identification Division.
 Program-ID. "inpcli".

 Data Division.
 Working-Storage Section.

* Copy Files for Constants and Structures.
 copy "sql.cbl".
 copy "sqlenv.cbl".
 copy "sqlca.cbl".

* Declare an Input/Output SQLDA Structure.
 01  io-sqlda sync.
     05 io-sqldaid     pic x(8) value "IN-DA   ".
     05 io-sqldabc     pic s9(9) comp-5.
     05 io-sqln        pic s9(4) comp-5.
     05 io-sqld        pic s9(4) comp-5.
     05 io-sqlvar-entries occurs 0 to 99 times
        depending on io-sqld.
        10 io-sqlvar.
           15 io-sqltype  pic s9(4) comp-5.
           15 io-sqllen   pic s9(4) comp-5.
           15 io-sqldata  usage is pointer.
           15 io-sqlind   usage is pointer.
           15 io-sqlname.
              20 io-sqlnamel   pic s9(4) comp-5.
              20 io-sqlnamec   pic x(30).

 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 dbname           pic x(8).
 01 userid           pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).

 01 table-name     pic x(10) value "PRESIDENTS".
 01 io-data1       pic x(20) value "Washington          ".
 01 io-data2       pic x(20) value "Jefferson           ".
 01 io-data3       pic x(20) value "Lincoln             ".

* Declare and Initialize Indicator Variables.
 01  table-nameind  pic s9(4) comp-5 value 0.
 01  io-dataind1    pic s9(4) comp-5 value 0.
 01  io-dataind2    pic s9(4) comp-5 value 0.
 01  io-dataind3    pic s9(4) comp-5 value 0.

 01  prog-name      pic x(12) value "inpsrv".
 EXEC SQL END DECLARE SECTION END-EXEC.


* Declare a Null Pointer Variable.
 77  null-ptr-int    pic s9(9)     comp-5.
 77  null-ptr redefines null-ptr-int pointer.

 77 errloc         pic x(80).
 Procedure Division.
* CONNECT TO DATABASE
     display "Enter in the database name : " with no advancing.
     accept dbname.

     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     move 0 to passwd-length.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.


* Call the Remote Procedure.
     display "Use CALL with Host Variable to invoke the Server Pro
-       "cedure".

     EXEC SQL CALL :prog-name  2a
                      (:table-name:table-nameind,
                      :io-data1:io-dataind1,
                      :io-data2:io-dataind2,
                      :io-data3:io-dataind3) END-EXEC.
     move "SQLCALL HV" to errloc.
     call "checkerr" using SQLCA errloc.
     display "Server Procedure Complete.".


*    Initialize the Input/Output SQLDA Structure
     move 4 to io-sqln.  1
     move 4 to io-sqld.

     move SQL-TYP-NCHAR to io-sqltype(1).
     set io-sqldata(1) to address of table-name.
     set io-sqlind(1)  to address of table-nameind.
     move 10 to io-sqllen(1).

     move SQL-TYP-NCHAR to io-sqltype(2).
     set io-sqldata(2) to address of io-data1.
     set io-sqlind(2)  to address of io-dataind1.
     move 20 to io-sqllen(2).

     move SQL-TYP-NCHAR to io-sqltype(3).
     set io-sqldata(3) to address of io-data2.
     set io-sqlind(3)  to address of io-dataind2.
     move 20 to io-sqllen(3).
     move SQL-TYP-NCHAR to io-sqltype(4).
     set io-sqldata(4) to address of io-data3.
     set io-sqlind(4)  to address of io-dataind3.
     move 20 to io-sqllen(4).

* Call the Remote Procedure.
     display "Use CALL with SQLDA to invoke the Server Procedure n
-       "amed".
     EXEC SQL CALL :prog-name USING DESCRIPTOR  2b
                              :io-sqlda END-EXEC.
     move "SQLCALL DA" to errloc.
     call "checkerr" using SQLCA errloc.
     display "Server Procedure Complete.".

* Disconnect from Remote Database.
     EXEC SQL CONNECT RESET END-EXEC.
     stop run.
     exit.
```

FORTRAN UNIX Example: INPCLI.SQF

```
      program inpcli
      implicit none

*     Copy Files for Constants and Structures
      include 'sql.f'
      include 'sqlenv.f'
      include 'sqlutil.f'
      include 'sqldact.f'

*     Declare an SQLCA
      EXEC SQL INCLUDE SQLCA

*     Declare host variables.
      EXEC SQL BEGIN DECLARE SECTION
        character*8    dbname
        character*8    userid
        character*18   passwd
        character*11   procname   /'inpsrv'/
        character*10   table_name /'PRESIDENTS'/
        character*20   data_item0 /'Washington'/
        character*20   data_item1 /'Jefferson'/
        character*20   data_item2 /'Lincoln'/
        integer*2      tableind  /0/
        integer*2      dataind0  /0/
        integer*2      dataind1  /0/
        integer*2      dataind2  /0/
      EXEC SQL END DECLARE SECTION


*     Declare Variables Used to Create an SQLDA
      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )
      integer*2  sqlvar2
      parameter ( sqlvar2 = sqlda_header_sz + 1*sqlvar_struct_sz )
      integer*2  sqlvar3
      parameter ( sqlvar3 = sqlda_header_sz + 2*sqlvar_struct_sz )
      integer*2  sqlvar4
      parameter ( sqlvar4 = sqlda_header_sz + 3*sqlvar_struct_sz )

*     Declare an Input SQLDA Structure -- 4 Variables
      character    io_sqlda(sqlda_header_sz + 4*sqlvar_struct_sz)

      character*8  io_sqldaid     ! Header
      integer*4    io_sqldabc
      integer*2    io_sqln
      integer*2    io_sqld

      integer*2    io_sqltype1    ! First Variable
      integer*2    io_sqllen1
      integer*4    io_sqldata1
      integer*4    io_sqlind1
      integer*2    io_sqlnamel1
      character*30 io_sqlnamec1

      integer*2    io_sqltype2    ! Second Variable
      integer*2    io_sqllen2
      integer*4    io_sqldata2
```

```
 integer*4    io_sqlind2
 integer*2    io_sqlnamel2
 character*30 io_sqlnamec2

 integer*2    io_sqltype3    ! Third Variable
 integer*2    io_sqllen3
 integer*4    io_sqldata3
 integer*4    io_sqlind3
 integer*2    io_sqlnamel3
 character*30 io_sqlnamec3

 integer*2    io_sqltype4    ! Fourth Variable
 integer*2    io_sqllen4
 integer*4    io_sqldata4
 integer*4    io_sqlind4
 integer*2    io_sqlnamel4
 character*30 io_sqlnamec4



 equivalence( io_sqlda(sqlda_sqldaid_ofs), io_sqldaid )
 equivalence( io_sqlda(sqlda_sqldabc_ofs), io_sqldabc )
 equivalence( io_sqlda(sqlda_sqln_ofs), io_sqln )
 equivalence( io_sqlda(sqlda_sqld_ofs), io_sqld )

 equivalence( io_sqlda(sqlvar1+sqlvar_type_ofs), io_sqltype1 )
 equivalence( io_sqlda(sqlvar1+sqlvar_len_ofs), io_sqllen1 )
 equivalence( io_sqlda(sqlvar1+sqlvar_data_ofs), io_sqldata1 )
 equivalence( io_sqlda(sqlvar1+sqlvar_ind_ofs), io_sqlind1 )
 equivalence( io_sqlda(sqlvar1+sqlvar_name_length_ofs),
+             io_sqlnamel1 )
 equivalence( io_sqlda(sqlvar1+sqlvar_name_data_ofs),
+             io_sqlnamec1 )

 equivalence( io_sqlda(sqlvar2+sqlvar_type_ofs), io_sqltype2 )
 equivalence( io_sqlda(sqlvar2+sqlvar_len_ofs), io_sqllen2 )
 equivalence( io_sqlda(sqlvar2+sqlvar_data_ofs), io_sqldata2 )
 equivalence( io_sqlda(sqlvar2+sqlvar_ind_ofs), io_sqlind2 )
 equivalence( io_sqlda(sqlvar2+sqlvar_name_length_ofs),
+             io_sqlnamel2 )
 equivalence( io_sqlda(sqlvar2+sqlvar_name_data_ofs),
+             io_sqlnamec2 )

 equivalence( io_sqlda(sqlvar3+sqlvar_type_ofs), io_sqltype3 )
 equivalence( io_sqlda(sqlvar3+sqlvar_len_ofs), io_sqllen3 )
 equivalence( io_sqlda(sqlvar3+sqlvar_data_ofs), io_sqldata3 )
 equivalence( io_sqlda(sqlvar3+sqlvar_ind_ofs), io_sqlind3 )
 equivalence( io_sqlda(sqlvar3+sqlvar_name_length_ofs),
+             io_sqlnamel3 )
 equivalence( io_sqlda(sqlvar3+sqlvar_name_data_ofs),
+             io_sqlnamec3 )

 equivalence( io_sqlda(sqlvar4+sqlvar_type_ofs), io_sqltype4 )
 equivalence( io_sqlda(sqlvar4+sqlvar_len_ofs), io_sqllen4 )
 equivalence( io_sqlda(sqlvar4+sqlvar_data_ofs), io_sqldata4 )
 equivalence( io_sqlda(sqlvar4+sqlvar_ind_ofs), io_sqlind4 )
 equivalence( io_sqlda(sqlvar4+sqlvar_name_length_ofs),
+             io_sqlnamel4 )
 equivalence( io_sqlda(sqlvar4+sqlvar_name_data_ofs),
     +             io_sqlnamec4 )

      character*80     errloc
      integer*4        rc

*     Program Logic
      print *, 'Enter in the database name :'
      read 100, dbname
100   format (a8)

      print *, 'Enter your user id (default none):'
      read 100, userid

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO :dbname
      else
      print *, 'Enter your password :'
      read 100, passwd

        print *,'CONNECT to Remote Database.'

      EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
      end if

*     Connect to Remote Database
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

*     Call the Remote Procedure via CALL with Host Variables  2
      print *,'Use CALL with Host Variable to invoke the ',
     + 'Server Procedure named inpsrv.'
      EXEC SQL CALL :procname (:table_name:tableind,
     +                         :data_item0:dataind0,
     +                         :data_item1:dataind1,
     +                         :data_item2:dataind2)
      errloc = 'CALL with HOST VARIABLES'
      call checkerr (sqlca, errloc, *999)
      print *,'Server Procedure Complete.'


*     Initialize the Input SQLDA Structure  1
      io_sqldaid   = 'IN_SQLDA'
      io_sqldabc   = sqlda_header_sz + 4*sqlvar_struct_sz
      io_sqln      = 4
      io_sqld      = 4

      io_sqltype1  = SQL_TYP_NCHAR
      io_sqllen1   = 10
      rc = sqlgaddr(%ref(table_name), %ref(io_sqldata1))
      rc = sqlgaddr(%ref(tableind), %ref(io_sqlind1))

      io_sqltype2  = SQL_TYP_NCHAR
      io_sqllen2   = 20
      rc = sqlgaddr(%ref(data_item0), %ref(io_sqldata2))
      rc = sqlgaddr(%ref(dataind0), %ref(io_sqlind2))

      io_sqltype3  = SQL_TYP_NCHAR
      io_sqllen3   = 20
      rc = sqlgaddr(%ref(data_item1), %ref(io_sqldata3))
      rc = sqlgaddr(%ref(dataind1), %ref(io_sqlind3))

      io_sqltype4  = SQL_TYP_NCHAR
      io_sqllen4   = 20
      rc = sqlgaddr(%ref(data_item2), %ref(io_sqldata4))
      rc = sqlgaddr(%ref(dataind2), %ref(io_sqlind4))

*     Call the Remote Procedure via CALL with SQLDA 2
      print *,'Use CALL with SQLDA to invoke the Server ',
     + 'Procedure named inpsrv.'
      EXEC SQL CALL :procname USING DESCRIPTOR :io_sqlda
      errloc = 'CALL with SQLDA'
      call checkerr (sqlca, errloc, *999)
      print *,'Server Procedure Complete.'

*     Disconnect from Remote Database.
      EXEC SQL CONNECT RESET
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)
999   stop
      end
```

REXX Example: INPCLI.CMD

```
/* REXX INPput CLIent  */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if rxfuncquery('SQLDBS') <> 0 then
     rcy = rxfuncadd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if rxfuncquery('SQLEXEC') <> 0 then
     rcy = rxfuncadd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")

/* pull in command line arguments */
parse arg dbname userid passwd .

/* check to see if the proper number of arguments have been passed in */
if passwd = '' then
do
  say 'USAGE : inpcli dbname userid passwd'
  exit
end

procname = 'inpsrv.cmd'
tablename = 'Presidents'
tableind = 0
dataitem.1 = 'Washington'
dataitem.1.ind = 0
dataitem.2 = 'Jefferson'
dataitem.2.ind = 0
dataitem.3 = 'Lincoln'
dataitem.3.ind = 0

io_sqlda.sqld = 4   1

io_sqlda.1.sqltype = 453
io_sqlda.1.sqldata = tablename
io_sqlda.1.sqllen  = 10
io_sqlda.1.sqlind  = tableind

io_sqlda.2.sqltype = 453
io_sqlda.2.sqldata = dataitem.1
io_sqlda.2.sqllen  = 20
io_sqlda.2.sqlind  = dataitem.1.ind

io_sqlda.3.sqltype = 453
io_sqlda.3.sqldata = dataitem.2
io_sqlda.3.sqllen  = 20
io_sqlda.3.sqlind  = dataitem.2.ind

io_sqlda.4.sqltype = 453
io_sqlda.4.sqldata = dataitem.3
io_sqlda.4.sqllen  = 20
io_sqlda.4.sqlind  = dataitem.3.ind
```

```
/* CONNECT TO REMOTE DATABASE */
say 'Connect to Remote Database.'
call SQLEXEC 'CONNECT TO 'dbname' USER 'userid' USING 'passwd
call CHECKERR 'CONNECT TO SAMPLE'

/* CALL THE REMOTE PROCEDURE USING SQL CALL AND HOST VARIABLES */
say 'Use CALL with Host Variables to invoke the Stored Procedure name inpsrv'
call SQLEXEC 'CALL :procname (:tablename:tableind,',     2a
             ':dataitem.1 :dataitem.1.ind,',
             ':dataitem.2 :dataitem.2.ind,',
             ':dataitem.3 :dataitem.3.ind)'
call CHECKERR 'CALL USING HV'
say 'Server Procedure Complete'
say

/* CALL THE REMOTE PROCEDURE USING SQL CALL AND SQLDA */
say 'Use CALL with SQLDA to invoke the Stored Procedure name inpsrv'
call SQLEXEC 'CALL :procname USING DESCRIPTOR :io_sqlda'  2b
call CHECKERR 'CALL USING SQLDA'
say 'Server Procedure Complete'


/* DISCONNECT FROM REMOTE DATABASE */
call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'
exit 0


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************\
    * GET ERROR MESSAGE API called *
    \******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

How the Example Input-SQLDA Stored Procedure Works

1. **Declare Server Procedure**. The procedure accepts pointers to SQLDA and SQLCA structures.
2. **Create Table**. Using the data passed in the first SQLVAR of the SQLDA structure, a CREATE TABLE statement is constructed and executed to create a table named `Presidents`.
3. **Prepare Insert Statement**. An INSERT statement with a parameter marker `?` is prepared.
4. **Insert Data**. The INSERT statement prepared previously is executed using the data passed in the second through fourth SQLVAR of the SQLDA structure. The parameter markers are replaced with the values `Washington`, `Jefferson`, and `Lincoln`. These values are inserted into the `Presidents` table.
5. **Return to the Client Application**. The server procedure copies the SQLCA to the SQLCA of the client application, issues a COMMIT statement if the transaction is successful, and returns the value `SQLZ_DISCONNECT_PROC`, indicating that no further calls to the server procedure will be made.

Note: Server procedures cannot be written in REXX on AIX systems.

C Example: INPSRV.SQC

```
#include <memory.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include "util.h"

SQL_API_RC SQL_API_FN inpsrv(void *reserved1,      1
           void *reserved2,
           struct sqlda   *inout_sqlda,
           struct sqlca   *ca)
{
   /* Declare a local SQLCA */
   EXEC SQL INCLUDE SQLCA;

   /* Declare Host Variables */
   EXEC SQL BEGIN DECLARE SECTION;
     char table_stmt[80]  = "CREATE TABLE ";
     char insert_stmt[80] = "INSERT INTO ";
     char insert_data[21];
   EXEC SQL END DECLARE SECTION;

   /* Declare Miscellanous Variables */
   int  cntr = 0;
   char *table_name;
   char *data_items[3];
   short  data_items_length[3];
   int  num_of_data = 0;

   /*-----------------------------------------------------------------*/
   /* Assign the data from the SQLDA to local variables so that we    */
   /* don't have to refer to the SQLDA structure further.  This will  */
   /* provide better portability to other platforms such as DB2 MVS   */
   /* where they receive the parameter list differently.              */
   /*-----------------------------------------------------------------*/

   table_name  = inout_sqlda->sqlvar[0].sqldata;
   num_of_data = inout_sqlda->sqld - 1;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      data_items[cntr] = inout_sqlda->sqlvar[cntr+1].sqldata;
      data_items_length[cntr] = inout_sqlda->sqlvar[cntr+1].sqllen;
   }

   /*-----------------------------------------------------------------*/
   /* Create President Table                                          */
   /* - For simplicity, we'll ignore any errors from the              */
   /*   CREATE TABLE so that you can run this program even when the   */
   /*   table already exists due to a previous run.                   */
   /*-----------------------------------------------------------------*/

   EXEC SQL WHENEVER SQLERROR CONTINUE;
   strcat(table_stmt, table_name);
   strcat(table_stmt, " (name CHAR(20))");

   EXEC SQL EXECUTE IMMEDIATE :table_stmt;  2
   EXEC SQL WHENEVER SQLERROR GOTO ext;

   /*-----------------------------------------------------------------*/
   /* Generate and execute a PREPARE for an INSERT statement, and     */
   /* then insert the three presidents.                               */
   /*-----------------------------------------------------------------*/

   strcat(insert_stmt, table_name);
   strcat(insert_stmt, "  VALUES (?)");   3

   EXEC SQL PREPARE S1 FROM :insert_stmt;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      strncpy(insert_data, data_items[cntr], data_items_length[cntr]);
      insert_data[data_items_length[cntr]] = '\0';
      EXEC SQL EXECUTE S1 USING :insert_data;   4
   }

   /*-----------------------------------------------------------------*/
   /* Return to caller                                                */
   /*   -  Copy the SQLCA                                             */
   /*   -  Update the output SQLDA.  Since there's no output to       */
   /*      return, we are setting the indicator values to -128 to     */
   /*      return only a null value.                                  */
   /*   -  Commit or Rollback the inserts.                            */
   /*-----------------------------------------------------------------*/

ext:   5
   memcpy(ca, &sqlca, sizeof(struct sqlca));
   if (inout_sqlda != NULL)
   {
     for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
     {
       *(inout_sqlda->sqlvar[cntr].sqlind) = -128;
     }
   }

   EXEC SQL WHENEVER SQLERROR CONTINUE;
   /* Check SQLCA for errors */
   if (SQLCODE == 0)
     EXEC SQL COMMIT;
   else
     EXEC SQL ROLLBACK;

   return(SQLZ_DISCONNECT_PROC);
}
```

COBOL Example: INPSRV.SQB

```
 Identification Division.
 Program-ID. "inpsrv".

 Data Division.
 Working-Storage Section.

* Copy Files for Constants and Structures
 copy "sql.cbl".
 copy "sqlenv.cbl".
 copy "sqlca.cbl".


* Declare Host Variables
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 insert-data       pic x(12).
   01 insert-statement  pic x(40).
 EXEC SQL END DECLARE SECTION END-EXEC.

* Declare and Initialize SQL Statement Strings
 01 create-string sync.
    05 stringc1         pic x(13) value "CREATE TABLE ".
    05 var-string1      pic x(10).
    05 stringc2         pic x(16) value " (NAME CHAR(20))".

 01 insert-string sync.
    05 stringi1         pic x(12) value "INSERT INTO ".
    05 var-string2      pic x(11).
    05 stringi2         pic x(11) value " VALUES (?)".

* Declare Miscellaneous Variables
 77 idx                 pic 9(4) comp-5.

 Linkage Section.

* Declare Parameters
 01 input-data.
    05 input-len        pic 9(4) comp-5.
    05 input-char       pic x(80).

 77 reserved1   pointer.
 77 reserved2   pointer.
 01  inout-sqlda sync.
     05 inout-sqldaid     pic x(8).
     05 inout-sqldabc     pic s9(9) comp-5.
     05 inout-sqln        pic s9(4) comp-5.
     05 inout-sqld        pic s9(4) comp-5.
     05 inout-sqlvar occurs 1 to 1489 times
        depending on inout-sqld.
        10 inout-sqltype  pic s9(4) comp-5.
        10 inout-sqllen   pic s9(4) comp-5.
        10 inout-sqldata  usage is pointer.
        10 inout-sqlind   usage is pointer.
        10 inout-sqlname.
           15 inout-sqlnamel   pic s9(4) comp-5.
           15 inout-sqlnamec   pic x(30).

* Declare Miscellaneous Variables
```

```
 77 temp-name           pic x(20).
 77 temp-ind            pic s9(4) comp-5.
 77 table-name          pic x(10).
 77 table-ind           pic s9(4) comp-5.

 01 O-SQLCA SYNC.
    05 SQLCAID PIC X(8).
    05 SQLCABC PIC S9(9) COMP-5.
    05 SQLCODE PIC S9(9) COMP-5.
    05 SQLERRM.
       49 SQLERRML PIC S9(4) COMP-5.
       49 SQLERRMC PIC X(70).
    05 SQLERRP PIC X(8).
    05 SQLERRD OCCURS 6 TIMES PIC S9(9) COMP-5.
    05 SQLWARN.
       10 SQLWARN0 PIC X.
       10 SQLWARN1 PIC X.
       10 SQLWARN2 PIC X.
       10 SQLWARN3 PIC X.
       10 SQLWARN4 PIC X.
       10 SQLWARN5 PIC X.
       10 SQLWARN6 PIC X.
       10 SQLWARN7 PIC X.
       10 SQLWARN8 PIC X.
       10 SQLWARN9 PIC X.
       10 SQLWARNA PIC X.
    05 SQLSTATE PIC X(5).


 Procedure Division using reserved1 reserved2 inout-sqlda O-SQLCA. 1

 Main Section.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

* Create "President" Table
     set address of table-name to inout-sqldata(1).
     move table-name to var-string1.
     move create-string to insert-statement.

     EXEC SQL EXECUTE IMMEDIATE :insert-statement END-EXEC.  2

     EXEC SQL WHENEVER SQLERROR GOTO :Error-Exit END-EXEC.

* Prepare for Insert
     move table-name to var-string2.
     move insert-string to insert-statement.
     EXEC SQL PREPARE INSERTSTMT FROM :insert-statement END-EXEC.  3

* Insert President Names stored in Input SQLDA into Newly Created Table
     perform Add-Rows varying idx from 2 by 1 until idx > 4.  4

* Return the SQLCA Information to the Calling Program.
     move SQLCA to O-SQLCA.
     EXEC SQL COMMIT END-EXEC.
     move SQLZ-DISCONNECT-PROC to return-code
     goback.  5

 Add-Rows.
     set address of temp-name to inout-sqldata(idx).
     move temp-name to insert-data.
     EXEC SQL EXECUTE INSERTSTMT USING :insert-data END-EXEC.

* To minimize network flow, set the input-only variable to null
* by setting its indicator value to -128 so that they won"t
* be resent back to the client program.
     set address of temp-ind  to inout-sqlind(idx).
     move -128      to temp-ind.

 Error-Exit.
* An Error has Occurred -- ROLLBACK and Return to Calling Program.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     move SQLCA to O-SQLCA.
     EXEC SQL ROLLBACK END-EXEC.
     move SQLZ-DISCONNECT-PROC to return-code
     goback.
```

FORTRAN UNIX Example: INPSRV.SQF

```
      integer*4 function inpsrv( reserved1, 1
     +                           reserved2,
     +                           io_sqlda,
     +                           ca )

      implicit none

*     Include Files for Constants and Structures
      include 'sql.f'
      include 'sqlenv.f'
      include 'sqlutil.f'
      include 'sqldact.f'

*     Declare Output SQLCA
      EXEC SQL INCLUDE SQLCA

*     Declare Dummy Parameters
      integer*4  reserved1
      integer*4  reserved2
      character  io_sqlda(sqlda_header_sz + 4*sqlvar_struct_sz)
      character  ca(sqlca_size)

*     Declare Host Variables
      EXEC SQL BEGIN DECLARE SECTION
        character*20   insert_data
*        character*80   insert_string


        character*12   sqlname        /' '/
        character*200  sql_string     /' '/
      EXEC SQL END DECLARE SECTION


*     Declare Local Variables to Hold Dummy Parameter Values
*     They are in_sqlda, out_sqlda and sqlca

*     Declare Variables Used to Create an SQLDA
      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )
      integer*2  sqlvar2
      parameter ( sqlvar2 = sqlda_header_sz + 1*sqlvar_struct_sz )
      integer*2  sqlvar3
      parameter ( sqlvar3 = sqlda_header_sz + 2*sqlvar_struct_sz )
      integer*2  sqlvar4
      parameter ( sqlvar4 = sqlda_header_sz + 3*sqlvar_struct_sz )

*     Declare an Input SQLDA Structure -- 4 Variables
      character   in_sqlda(sqlda_header_sz + 4*sqlvar_struct_sz)

      character*8  in_sqldaid     ! Header
      integer*4    in_sqldabc
      integer*2    in_sqln
      integer*2    in_sqld

      integer*2    in_sqltype1    ! First Variable
      integer*2    in_sqllen1
      integer*4    in_sqldata1
      integer*4    in_sqlind1
      integer*2    in_sqlnamel1
      character*30 in_sqlnamec1

      integer*2    in_sqltype2    ! Second Variable
      integer*2    in_sqllen2
      integer*4    in_sqldata2
      integer*4    in_sqlind2
      integer*2    in_sqlnamel2
      character*30 in_sqlnamec2

      integer*2    in_sqltype3    ! Third Variable
      integer*2    in_sqllen3
      integer*4    in_sqldata3
      integer*4    in_sqlind3
      integer*2    in_sqlnamel3
      character*30 in_sqlnamec3

      integer*2    in_sqltype4    ! Fourth Variable
      integer*2    in_sqllen4
      integer*4    in_sqldata4
      integer*4    in_sqlind4
      integer*2    in_sqlnamel4
      character*30 in_sqlnamec4


      equivalence( in_sqlda(sqlda_sqldaid_ofs),        in_sqldaid   )
      equivalence( in_sqlda(sqlda_sqldabc_ofs),        in_sqldabc   )
      equivalence( in_sqlda(sqlda_sqln_ofs),           in_sqln      )
      equivalence( in_sqlda(sqlda_sqld_ofs),           in_sqld      )

      equivalence( in_sqlda(sqlvar1+sqlvar_type_ofs),  in_sqltype1  )
      equivalence( in_sqlda(sqlvar1+sqlvar_len_ofs),   in_sqllen1   )
      equivalence( in_sqlda(sqlvar1+sqlvar_data_ofs),  in_sqldata1  )
      equivalence( in_sqlda(sqlvar1+sqlvar_ind_ofs),   in_sqlind1   )
      equivalence( in_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             in_sqlnamel1 )
      equivalence( in_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             in_sqlnamec1 )

      equivalence( in_sqlda(sqlvar2+sqlvar_type_ofs),  in_sqltype2  )
      equivalence( in_sqlda(sqlvar2+sqlvar_len_ofs),   in_sqllen2   )
      equivalence( in_sqlda(sqlvar2+sqlvar_data_ofs),  in_sqldata2  )
      equivalence( in_sqlda(sqlvar2+sqlvar_ind_ofs),   in_sqlind2   )
      equivalence( in_sqlda(sqlvar2+sqlvar_name_length_ofs),
     +             in_sqlnamel2 )
      equivalence( in_sqlda(sqlvar2+sqlvar_name_data_ofs),
     +             in_sqlnamec2 )

      equivalence( in_sqlda(sqlvar3+sqlvar_type_ofs),  in_sqltype3  )
      equivalence( in_sqlda(sqlvar3+sqlvar_len_ofs),   in_sqllen3   )
      equivalence( in_sqlda(sqlvar3+sqlvar_data_ofs),  in_sqldata3  )
      equivalence( in_sqlda(sqlvar3+sqlvar_ind_ofs),   in_sqlind3   )
      equivalence( in_sqlda(sqlvar3+sqlvar_name_length_ofs),
     +             in_sqlnamel3 )
      equivalence( in_sqlda(sqlvar3+sqlvar_name_data_ofs),
     +             in_sqlnamec3 )

      equivalence( in_sqlda(sqlvar4+sqlvar_type_ofs),  in_sqltype4  )
      equivalence( in_sqlda(sqlvar4+sqlvar_len_ofs),   in_sqllen4   )
      equivalence( in_sqlda(sqlvar4+sqlvar_data_ofs),  in_sqldata4  )
      equivalence( in_sqlda(sqlvar4+sqlvar_ind_ofs),   in_sqlind4   )
      equivalence( in_sqlda(sqlvar4+sqlvar_name_length_ofs),
     +             in_sqlnamel4 )
      equivalence( in_sqlda(sqlvar4+sqlvar_name_data_ofs),
     +             in_sqlnamec4 )

*     Declare and Initialize SQL Statement Strings
      character    create_string(39)
      character*39 create_string_tmp
      character*13 stringc1          /'CREATE TABLE '/
      character*10 var_string1
      character*16 stringc2          /' (NAME CHAR(20))'/

      equivalence( create_string(1),  stringc1          )
      equivalence( create_string(14), var_string1       )
      equivalence( create_string(24), stringc2          )
      equivalence( create_string(1),  create_string_tmp )

      character    insert_string(34)
      character*34 insert_string_tmp
      character*12 stringi1          /'INSERT INTO '/
      character*10 var_string2
      character*11 stringi2          /' VALUES (?)'/

      equivalence( insert_string(1),  stringi1          )
      equivalence( insert_string(13), var_string2       )
      equivalence( insert_string(23), stringi2          )
      equivalence( insert_string(1),  insert_string_tmp )

*     Declare Miscellaneous Variables
      integer*2    cntr
      integer*2    rc

*     Copy io_sqlda to in_sqlda
      cntr = 0
      do 20 while ( cntr .lt. (sqlda_header_sz + 4*sqlvar_struct_sz) )
        cntr = cntr + 1
        in_sqlda(cntr) = io_sqlda(cntr)
20    end do

*     Program Logic

      EXEC SQL WHENEVER SQLERROR CONTINUE

*     Create "President" Table
      rc = sqlgdref ( %val(in_sqllen1), %ref(var_string1),
     +     %ref(in_sqldata1) )
      sql_string = create_string_tmp
      EXEC SQL EXECUTE IMMEDIATE :sql_string  2

      EXEC SQL WHENEVER SQLERROR GOTO 100

*     Prepare for Insert
      var_string2 = var_string1
      sql_string = insert_string_tmp
      EXEC SQL PREPARE insertstmt FROM :sql_string  3
```

```
*     Insert President Names stored in Input SQLDA into Newly Created Table
      rc = sqlgdref( %val(in_sqllen2),%ref(insert_data),
     +     %ref(in_sqldata2) )
      EXEC SQL EXECUTE insertstmt USING :insert_data 4
      rc = sqlgdref( %val(in_sqllen3),%ref(insert_data),
     +     %ref(in_sqldata3) )
      EXEC SQL EXECUTE insertstmt USING :insert_data
      rc = sqlgdref( %val(in_sqllen4),%ref(insert_data),
     +     %ref(in_sqldata4) )
      EXEC SQL EXECUTE insertstmt USING :insert_data

      EXEC SQL COMMIT

*     Copy Local Variables into Dummy Output Variables
*     Copy sqlca to ca
      cntr = 0
      do 30 while ( cntr .lt. sqlca_size )
        cntr = cntr + 1
        ca(cntr) = sqlca(cntr)
30    end do

      goto 200

100   continue    ! An Error has Occurred
*     Copy Local Variables into Dummy Output Variables
*     Copy sqlca to ca
      EXEC SQL WHENEVER SQLERROR CONTINUE
      cntr = 0
      do 40 while ( cntr .lt. sqlca_size )
        cntr = cntr + 1
        ca(cntr) = sqlca(cntr)
40    end do

      EXEC SQL ROLLBACK

200   continue                                 ! Exit Program
      inpsrv = sqlz_disconnect_proc
      return 5
      end
```

REXX OS/2 Example: INPSRV.CMD

```
/* REXX INPut SeRVer */

/* this variable (SYSTEM) must be user defined */
SYSTEM = OS2
if SYSTEM = OS2 then do
  if rxfuncquery('SQLDBS') <> 0 then
    rcy = rxfuncadd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )

  if rxfuncquery('SQLEXEC') <> 0 then
    rcy = rxfuncadd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rexx")


table_stmt = 'CREATE TABLE ' || sqlrida.1.sqldata ||  ' (name CHAR(20))'  2

call SQLEXEC 'EXECUTE IMMEDIATE :table_stmt'
if sqlca.sqlcode = -601 then
  sqlca.sqlcode = 0
call CHECKERR 'EXECUTE IMMEDIATE CREATE TABLE'

insert_stmt = 'INSERT INTO ' || sqlrida.1.sqldata || ' VALUES (?)'
call SQLEXEC 'PREPARE s1 FROM :insert_stmt'   3
call CHECKERR 'PREPARE INSERT STATEMENT'

cntr = 1
do while ( cntr < SQLRIDA.sqld & sqlca.sqlcode = 0 )
   cntr        = cntr + 1
   insert_data = SQLRIDA.cntr.sqldata
   call SQLEXEC 'EXECUTE s1 USING :insert_data'   4
end
call CHECKERR 'EXECUTION OF INSERT STATEMENT'

call SQLEXEC 'COMMIT'
call CHECKERR 'COMMIT'

exit 0   5


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else
    say '--- error report ---'
    say 'ERROR occured :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****************************\
    * GET ERROR MESSAGE API called *
    \*****************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'
    if (SQLCA.SQLCODE < 0 ) then
      exit
    else
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Registering Stored Procedures

Unlike other objects such as tables, indexes, functions, and triggers, there is no table or view in the catalog for stored procedures. However, a special table, DB2CLI.PROCEDURES (a pseudo-catalog table), has been defined by DB2 that lists and describes available stored procedures, along with the associated parameters of those stored procedures. It is recommended that administrators insert entries into this table for all stored procedures in the database. This table, and the steps for creating this table are described in the appendix of the *CLI Guide and Reference*.

Returning Result Sets From Stored Procedures

Stored procedures invoked through DB2 CLI provide a capability not available with those invoked through embedded SQL; that is, the ability to return one or more result sets to the client application. Aspects of this support include:

- The only client API provided is via CLI.
- The client application program can describe the result sets returned.
- Result sets must be processed in serial fashion by the application. A cursor is automatically opened on the first result set and a special call (SQLMoreResults) is provided to both close the cursor on one result set and to open it on the next.
- The stored procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on that result set, and leaving the cursor open when exiting the procedure. If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened (passing back any unread rows).
- The RESULT_SETS column in the DB2CLI.PROCEDURES table indicates whether or not a stored procedure can return result sets. Only procedures with a value greater than zero in this column will return result sets for open cursors.

For additional details, see the section on *Using Stored Procedures* in the *CLI Guide and Reference*.

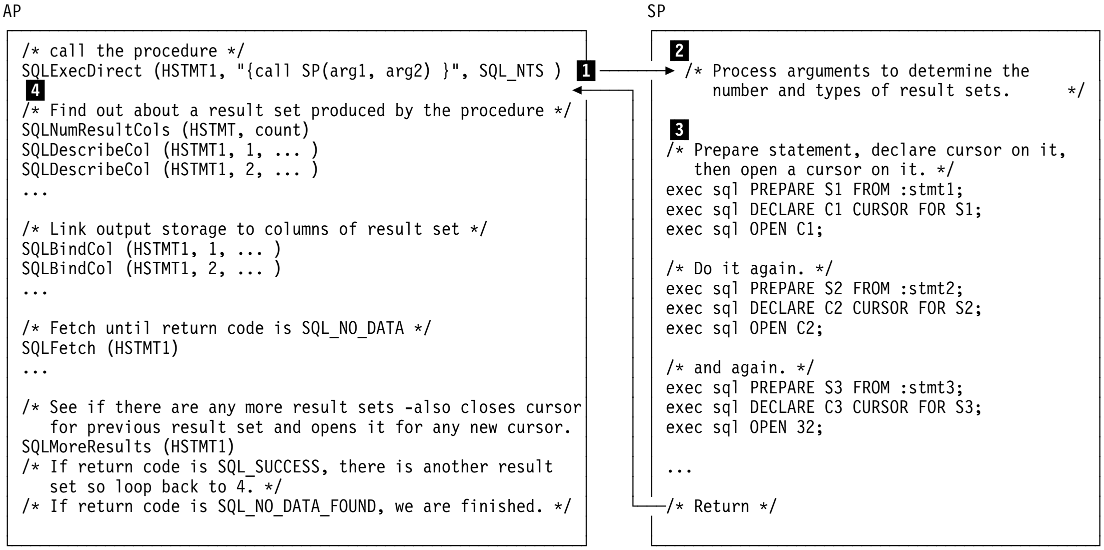
The diagram below shows typical use of this facility:

1 A client application program AP calls a procedure SP providing two input arguments.

2 SP determines, based on those arguments, that 3 result set are to be returned.

3 SP provides open cursors on the statements that define the result sets.

4 AP processes the result sets in a general loop that is able to deal with different numbers and different types of results.



```
/* call the procedure */
SQLExecDirect (HSTMT1, "{call SP(arg1, arg2) }", SQL_NTS ) 1
 4
/* Find out about a result set produced by the procedure */
SQLNumResultCols (HSTMT, count)
SQLDescribeCol (HSTMT1, 1, ... )
SQLDescribeCol (HSTMT1, 2, ... )
...

/* Link output storage to columns of result set */
SQLBindCol (HSTMT1, 1, ... )
SQLBindCol (HSTMT1, 2, ... )
...

/* Fetch until return code is SQL_NO_DATA */
SQLFetch (HSTMT1)
...

/* See if there are any more result sets -also closes cursor
   for previous result set and opens it for any new cursor.
SQLMoreResults (HSTMT1)
/* If return code is SQL_SUCCESS, there is another result
   set so loop back to 4. */
/* If return code is SQL_NO_DATA_FOUND, we are finished. */
```

```
 2
 /* Process arguments to determine the
    number and types of result sets.      */

 3
/* Prepare statement, declare cursor on it,
   then open a cursor on it. */
exec sql PREPARE S1 FROM :stmt1;
exec sql DECLARE C1 CURSOR FOR S1;
exec sql OPEN C1;

/* Do it again. */
exec sql PREPARE S2 FROM :stmt2;
exec sql DECLARE C2 CURSOR FOR S2;
exec sql OPEN C2;

/* and again. */
exec sql PREPARE S3 FROM :stmt3;
exec sql DECLARE C3 CURSOR FOR S3;
exec sql OPEN 32;

...

/* Return */
```

Note that this ability to return one or more result sets to a client application is available to DB2 Universal Database clients using embedded SQL if the stored procedure resides on a server that is accessible from a DataJoiner Version 2 server. This capability may also be available on DRDA host platforms. Consult the product documentation for DataJoiner or for the DB2 DRDA host platform for more information.

Chapter 6. Using the Object-Relational Capabilities

This chapter discusses the object-oriented capabilities of DB2 and how you use them. Topics discussed include:

- Why Use the DB2 Object Extensions?
- DB2 Approach to Supporting Objects
- Using Large Objects (LOBs)
- User-Defined Functions (UDF)
- User-defined Distinct Types (UDT)
- Synergy Between UDTs, UDFs, and LOBs

One of the most important recent developments in modern programming language technology is *object-orientation*. Object orientation is the notion that entities in the application domain can be modeled as independent objects that are related to one another by means of classification. Object-orientation lets you capture the similarities and differences among objects in your application domain and group those objects together into related types. Objects of the same type behave in the same way because they share the same set of type-specific functions, reflecting the behavior of your objects in the application domain.

Why Use the DB2 Object Extensions?

With the object extensions of DB2, you can incorporate object-oriented (OO) concepts and methodologies into your relational database by extending it with richer sets of types and functions. With these extensions, you can store instances of object-oriented data types in columns of tables, and operate on them by means of functions in SQL statements. In addition, you can make the semantic behavior of stored objects an important resource that can be shared among all your applications by means of the database.

To incorporate object-orientation into your relational database systems, you can define new types and functions of your own to reflect the semantics of the objects in your application domain. Some of the data objects you want to model may be large and complex (for example, text, voice, image, and financial data). Therefore, you may also need mechanisms for the storage and manipulation of large objects. User-defined Distinct types (UDTs), user-defined functions (UDFs), and Large Objects (LOBs) are the mechanisms provided by DB2 to fulfill these needs. With DB2, you can now define new types and functions of your own to store and manipulate application objects, potentially gigabytes in size, within the database.

UDTs, UDFs, and LOBs build the foundation for the DB2 object-relational extensions. In addition to UDTs, UDFs, and LOBs, DB2 also includes mechanisms that enable the representation of another important piece of the object semantics: the *rules* that govern the kinds of operations that are allowed on such objects in the application domain. These rules are embodied in the DB2 triggers and constraints mechanisms.

As described in subsequent sections, there is an important synergy among these object-oriented features. You can model a complex object in the application domain as a UDT. The UDT may in turn be internally represented as a LOB. The UDT's behavior may be implemented in terms of UDFs, and its integrity rules implemented in terms of constraints and triggers. This section shows you the steps required to define UDTs and UDFs, how to use LOBs, and how UDTs, UDFs, and LOBs can better represent the objects in your application and thus work together. Chapter 8, "Using the Active DBMS Capabilities" on page 349 shows you how triggers can enhance the semantics of your applications.

Note: The use of the DB2 object-oriented mechanisms (UDTs, UDFs, LOBs, and triggers) is not restricted to the support of object-oriented applications. Just as C++, a popular object-oriented programming language, is used to implement all sorts of non-object-oriented applications, the object-oriented mechanisms provided by DB2 are also very useful to support all kinds of non-object-oriented applications. UDTs, UDFs, LOBs, triggers, and constraints are general-purpose mechanisms that can be used to model any database application. For this reason, these DB2 object extensions offer extensive support for both non-traditional, that is, object-oriented applications, in addition to improving support for traditional ones.

DB2 Approach to Supporting Objects

The object extensions of DB2 enable you to realize many of the benefits of object technology while building on the strengths of relational technology. In a relational system, data types are used to describe the data stored in columns of tables where the instances (or objects) of these data types are stored. Operations on these instances are supported by means of operators or functions that can be invoked anywhere that expressions are allowed.

The DB2 approach to support object extensions fits exactly into the relational paradigm. UDTs are data types defined by the user which, like built-in types, can be used to describe the data stored in columns of tables. UDFs are functions defined by a user which, like built-in functions or operators, support the manipulation of UDT instances. Thus, UDT instances are stored in columns of tables and manipulated by UDFs in SQL queries. UDTs can be internally represented in different ways. LOBs are just one example of this.

Using Large Objects (LOBs)

The LONG VARCHAR and LONG VARGRAPHIC data types have a limit of 32K bytes of storage. While this may be sufficient for small to medium size text data, applications often need to store large text documents. They may 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 two (2) gigabytes (GB) in size. The three data types are: Binary Large OBjects (BLOBs), single-byte Character Large OBjects (CLOBs), and Double-Byte Character Large OBjects (DBCLOBs).

Along with storing large objects (LOBs), a way is also needed to refer to, and to use and modify, each LOB in the database. Each DB2 table may have a large amount of associated LOB data. Although any single LOB value may not exceed 2 gigabytes, a single row may contain as much as 24 gigabytes of LOB data, and a table may contain as much as 4 terabytes of LOB data. The content of the LOB column of a particular row at any point in time has *a large object value*.

You can refer to and manipulate LOBs using host variables just as you would any other data type. However, host variables use the client memory buffer which may not be large enough to hold LOB values. Other means are necessary to manipulate these large values. *Locators* are useful to identify and manipulate a large object value at the database server and for extracting pieces of the LOB value. *File reference variables* are useful for physically moving a large object value (or a large part of it) to and from the client.

The subsections that follow discuss in more detail those topics introduced above.

Understanding Large Object Data Types (BLOB, CLOB, DBCLOB)

Large object data types store data ranging in size from zero bytes to two gigabytes - 1.

The three large object data types have the following definitions:

- Character Large OBjects (CLOBs) — A character string made up of single-byte characters with an associated code page. This data type is best for holding text-oriented information where the amount of information could grow beyond the limits of a regular VARCHAR data type (upper limit of 4K bytes). Code page conversion of the information is supported as well as compatibility with the other character types.
- Double-Byte Character Large OBjects (DBCLOBs) — A character string made up of double-byte characters with an associated code page. This data type is best for holding text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported as well as compatibility with the other character types.
- Binary Large OBjects (BLOBs) — A binary string made up of bytes with no associated code page. This data type may be the most useful because it can store binary data, making it a perfect source type for use by User-defined Distinct Types (UDTs). UDTs using BLOBs as the source type are created to store image, voice, graphical, and other types of business or application specific data. For more information on UDTs, see “User-defined Distinct Types (UDT)” on page 271.

A separate database location stores all large object values outside their records in the table. There is a *large object descriptor* for each large object in each row in a table. The large object descriptor contains control information used to access the large object data stored elsewhere on disk. It is the storing of large object data outside their records that allows LOBs to be 2 GB in size. Accessing the large object descriptor causes a small amount of overhead when manipulating LOBs. (For storage and performance reasons you would likely not want to put small data items into LOBs.)

The maximum size for each large object column is part of the declaration of the large object type in the CREATE TABLE statement. The maximum size of a large object column determines the maximum size of any LOB descriptor in that column. As a result, it also determines how many columns of all data types can fit in a single row. The space used by the LOB descriptor in the row ranges from approximately 60 to 300 bytes, depending on the maximum size of the corresponding column. For specific sizes of the LOB descriptor, see the `CREATE TABLE` statement in the *SQL Reference*.

The `lob-options-clause` on CREATE TABLE gives you the choice to log (or not) the changes made to the LOB column(s). This clause also allows for a compact representation for the LOB descriptor (or not). This means you can allocate only enough space to store the LOB or you can allocate extra space for future append operations to the LOB. The `tablespace-options-clause` allows you to identify a LONG table space to store the column values of long field or LOB data types. For more information on the `CREATE TABLE` and `ALTER TABLE` statements, see the *SQL Reference*.

With their potentially very large size, LOBs can slow down the performance of your database system significantly when moved into or out of a database. Even though DB2 does not allow logging of a LOB value greater than 1 GB, LOB values with sizes near several hundred megabytes can quickly push the database log to near capacity. An error, SQLCODE -355 (SQLSTATE 42993), results from attempting to log a LOB greater than 1 GB in size. The `lob-options-clause` in the `CREATE TABLE` and `ALTER TABLE` statements allows users to turn off logging for a particular LOB column. Although setting the option to `NOT LOGGED` improves performance, changes to the LOB values after the most recent backup are lost during roll-forward recovery. For more information on these topics, see the *Administration Guide*.

Understanding Large Object Locators

Conceptually, LOB locators represent a simple idea that has been around for a while; use a small, easily managed value to refer to a much larger value. Specifically, a LOB locator is a 4-byte value stored in a host variable that a program can use to refer to a LOB value (or LOB expression) held in the database system. Using a LOB locator, a program can manipulate the LOB value as if the LOB value was stored in a regular host variable. The difference in using the LOB locator is that there is no need to transport the LOB value from the server to the application (and possibly back again).

The LOB locator is associated with a LOB value or LOB expression, not a row or physical storage location in the database. Therefore, after selecting a LOB value into a locator, there is no operation that you could perform on the original row(s) or tables(s) that would have any effect on the value referenced by the locator. The value associated with the locator is valid until the unit of work ends, or the locator is explicitly freed, whichever comes first. The `FREE LOCATOR` statement releases a locator from its associated value. In a similar way, a commit or roll-back operation frees all LOB locators associated with the transaction.

LOB locators can also be passed between DB2 and UDFs. There are special APIs available for UDFs to manipulate the LOB values using LOB locators. For more information on these APIs see “Using LOB Locators as UDF Parameters or Results” on page 315.

When selecting a LOB value, you have three options:

- Select the entire LOB value into a host variable. The entire LOB value is copied from the server to the client.
- Select just a LOB locator into a host variable. The LOB value remains on the server; the LOB locator moves to the client.
- Select the entire LOB value into a file reference variable. The LOB value is moved to a file at the client without going through the application's memory.

The use of the LOB value within the program can help the programmer determine which method is best. If the LOB value is very large and is needed only as an input value for one or more subsequent SQL statements, then it is best to keep the value in a locator. The use of a locator eliminates any client/server communication traffic needed to transfer the LOB value to the host variable and back to the server.

If the program needs the entire LOB value regardless of the size, then there is no choice but to transfer the LOB. Even in this case, there are still three options available to you. You can select the entire value into a regular or file host variable, but it may also work out better to select the LOB value into a locator and read it piecemeal from the locator into a regular host variable, as suggested in the following example.

Example: Using a Locator to Work With a CLOB Value

In this example, the application program retrieves a locator for a LOB value; then it uses the locator to extract the data from the LOB value. Using this method, the program allocates only enough storage for one piece of LOB data (the size is determined by the program) and it needs to issue only one fetch call using the cursor.

How the Sample LOBLOC Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. CLOB LOCATOR host variables are declared.
2. **Fetch the LOB value into the host variable LOCATOR.** A CURSOR and FETCH routine is used to obtain the location of a LOB field in the database to a host variable locator.
3. **Free the LOB LOCATORS.** The LOB LOCATORS used in this example are freed, releasing the locators from their previously associated values.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl.

FORTRAN CHECKERR is a subroutine located in the util.f file.

See "Using GET ERROR MESSAGE in Example Programs" on page 85 for the source code for this error checking utility.

C Sample: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

#ifdef DB2MAC
   char * bufptr;
#endif

   EXEC SQL BEGIN DECLARE SECTION;  1
      char number[7];
      long deptInfoBeginLoc;
      long deptInfoEndLoc;
      SQL TYPE IS CLOB_LOCATOR resume;
      SQL TYPE IS CLOB_LOCATOR deptBuffer;
      short lobind;
      char buffer[1000]="";
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBLOC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobloc [userid passwd]\n\n");
      return 1;
   } /* endif */

   /* Employee A10030 is not included in the following select, because
      the lobeval program manipulates the record for A10030 so that it is
      not compatible with lobloc */

   EXEC SQL DECLARE c1 CURSOR FOR
            SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
            AND empno <> 'A00130';

   EXEC SQL OPEN c1;
   CHECKERR ("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :number, :resume :lobind;   2
      if (SQLCODE != 0) break;
      if (lobind < 0) {
         printf ("NULL LOB indicated\n");
      } else {
         /* EVALUATE the LOB LOCATOR */
         /* Locate the beginning of "Department Information" section */
         EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
            INTO :deptInfoBeginLoc;
         CHECKERR ("VALUES1");

         /* Locate the beginning of "Education" section (end of "Dept.Info" */
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
         CHECKERR ("VALUES2");

         /* Obtain ONLY the "Department Information" section by using SUBSTR */
         EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
         CHECKERR ("VALUES3");

         /* Append the "Department Information" section to the :buffer var. */
         EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
         CHECKERR ("VALUES4");
      } /* endif */
   } while ( 1 );

#ifdef DB2MAC
   /* Need to convert the newline character for the Mac */
   bufptr = &(buffer[0]);
   while ( *bufptr != '\0' ) {
      if ( *bufptr == 0x0A ) *bufptr = 0x0D;
        bufptr++;
   }
#endif

   printf ("%s\n",buffer);

   EXEC SQL FREE LOCATOR :resume, :deptBuffer;  3
   CHECKERR ("FREE LOCATOR");

   EXEC SQL CLOSE c1;
   CHECKERR ("CLOSE CURSOR");

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : LOBLOC.SQC */
```

COBOL Sample: LOBLOC.SQB

```
 Identification Division.
 Program-ID. "lobloc".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 empnum            pic x(6).
 01 di-begin-loc      pic s9(9) comp-5.
 01 di-end-loc        pic s9(9) comp-5.
 01 resume            USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 di-buffer         USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 lobind            pic s9(4) comp-5.
 01 buffer            USAGE IS SQL TYPE IS CLOB(1K).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBLOC".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that it is
* not compatible with lobloc

     EXEC SQL DECLARE c1 CURSOR FOR
              SELECT empno, resume FROM emp_resume
              WHERE resume_format = 'ascii'
              AND empno <> 'A00130' END-EXEC.

     EXEC SQL OPEN c1 END-EXEC.
     move "OPEN CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     Move 0 to buffer-length.

     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

* display contents of the buffer.
     display buffer-data(1:buffer-length).

     EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC.  3
     move "FREE LOCATOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CLOSE c1 END-EXEC.
     move "CLOSE CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.
 End-Main.
        go to End-Prog.

 Fetch-Loop Section.
     EXEC SQL FETCH c1 INTO :empnum, :resume :lobind  2
        END-EXEC.

     if SQLCODE not equal 0
        go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
     if lobind less than 0 go to NULL-lob-indicated.

* Value exists.  Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
     EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
              INTO :di-begin-loc END-EXEC.
     move "VALUES1" to errloc.
     call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
     EXEC SQL VALUES (POSSTR(:resume, 'Education'))
               INTO :di-end-loc END-EXEC.
     move "VALUES2" to errloc.
     call "checkerr" using SQLCA errloc.

     subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
     EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
              :di-end-loc))
              INTO :di-buffer END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
* Append the "Department Information" section to the :buffer var
     EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
              END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

     go to End-Fetch-Loop.

 NULL-lob-indicated.
     display "NULL LOB indicated".

 End-Fetch-Loop. exit.

 End-Prog.
           stop run.
```

FORTRAN Sample: LOBLOC.SQF

```
      program lobloc
      implicit none

      include 'sqlenv.f'
      include 'sql.f'
      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION 1
        character*8              userid
        character*18             passwd
        character*6              empnum
        integer*4                di_begin_loc
        integer*4                di_end_loc
        SQL TYPE IS CLOB_LOCATOR resume
        SQL TYPE IS CLOB_LOCATOR di_buffer
        integer*2                lobind
        SQL TYPE IS CLOB(1K)     buffer
      EXEC SQL END DECLARE SECTION

      character*80     errloc

      print *, 'Sample Fortran Program: LOBLOC'

      print *, 'Enter your user id (default none):'
      read 101, userid
101   format (a8)

      if( userid(1:1) .eq. ' ' ) then
      EXEC SQL CONNECT TO sample
      else
      print *, 'Enter your password :'
      read 101, passwd

      EXEC SQL CONNECT TO sample USER :userid USING :passwd
      end if
      errloc = 'CONNECT'
      call checkerr (sqlca, errloc, *999)

*     Employee A10030 is not included in the following select, because
*     the lobeval program manipulates the record for A10030 so that it is
*     not compatible with lobloc

      EXEC SQL DECLARE c1 CURSOR FOR
     +         SELECT empno, resume FROM emp_resume
     +         WHERE resume_format = 'ascii'
     +         AND empno <> 'A00130'

      EXEC SQL OPEN c1
      errloc = 'OPEN CURSOR'
      call checkerr (sqlca, errloc, *999)


   10 continue
         EXEC SQL FETCH c1 INTO :empnum, :resume :lobind 2
         if (sqlcode .ne. 0) goto 100
         if (lobind .lt. 0) goto 50
```

```
*     The CLOB value exists.
*     Locate the beginning of the "Department Information" section
         EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
     +           INTO :di_begin_loc
         errloc = 'VALUES1'
         call checkerr (sqlca, errloc, *999)

*     Locate the beginning of the "Education" section (end of Dept.Info)
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
     +           INTO :di_end_loc
         errloc = 'VALUES2'
         call checkerr (sqlca, errloc, *999)

*     Obtain ONLY the "Department Information" section by using SUBSTR
         EXEC SQL VALUES(SUBSTR(:resume, :di_begin_loc,
     +           :di_end_loc - :di_begin_loc)) INTO :di_buffer
         errloc = 'VALUES3'
         call checkerr (sqlca, errloc, *999)

*     Append the "Department Information" section to the :buffer variable
         EXEC SQL VALUES(:buffer || :di_buffer) INTO :buffer
         errloc = 'VALUES4'
         call checkerr (sqlca, errloc, *999)

      goto 10

   50 print *,'NULL LOB indicated'
      goto 10

  100 print *, buffer

      EXEC SQL FREE LOCATOR :resume, :di_buffer  3
      errloc = 'FREE LOCATOR'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL CLOSE c1
      errloc = 'CLOSE CURSOR'
      call checkerr (sqlca, errloc, *999)

      EXEC SQL CONNECT RESET
      errloc = 'CONNECT RESET'
      call checkerr (sqlca, errloc, *999)

  999 stop
      end
```

Example: Deferring the Evaluation of a LOB Expression

There is no movement of the bytes of a LOB value until the assignment of a LOB expression to a target destination. This means that a LOB value locator used with string functions and operators can create an expression where the evaluation is postponed until the time of assignment. This is called *deferring evaluation* of a LOB expression.

In this example, a particular resume (`empno = '000130'`) is sought within a table of resumes `EMP_RESUME`. The `Department Information` section of the resume is copied, cut, and then appended to the end of the resume. This new resume will then be inserted into the `EMP_RESUME` table. The original resume in this table remains unchanged.

Locators permit the assembly and examination of the new resume without actually moving or copying any bytes from the original resume. The movement of bytes does not happen until the final assignment; that is, the `INSERT` statement — and then only at the server.

Deferring evaluation gives DB2 an opportunity to increase LOB I/O performance. This occurs because the LOB function optimizer attempts to transform the LOB expressions into alternative expressions. These alternative expressions produce equivalent results but may also require fewer disk I/Os.

In summary, LOB locators are ideally suited for a number of programming scenarios:

1. When moving only a small part of a much larger LOB to a client program.
2. When the entire LOB cannot fit in the application’s memory.
3. When the program needs a temporary LOB value from a LOB expression but does not need to save the result.
4. When performance is important (by deferring evaluation of LOB expressions).

How the Sample LOBEVAL Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. CLOB LOCATOR host variables are declared.
2. **Fetch the LOB value into the host variable LOCATOR.** A CURSOR and FETCH routine is used to obtain the location of a LOB field in the database to a host variable locator.
3. **LOB data is manipulated through the use of LOCATORS.** The next five SQL statements manipulate the LOB data without moving the actual data contained in the LOB field. This is done through the use of the LOB LOCATORS.
4. **LOB data is moved to the target destination.** The evaluation of the LOB assigned to the target destination is postponed until this SQL statement. The evaluation of this LOB statement has been deferred.
5. **Free the LOB LOCATORS.** The LOB LOCATORS used in this example are freed, releasing the locators from their previously associated values.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cbl.

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Sample: LOBEVAL.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      char userid[9];
      char passwd[19];
      long            hv_start_deptinfo;
      long            hv_start_educ;
      long            hv_return_code;
      SQL TYPE IS CLOB(5K) hv_new_section_buffer;
      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;

   printf( "Sample C program: LOBEVAL\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobeval [userid passwd]\n\n");
      return 1;
   } /* endif */

   /* delete any instance of "A00130" from previous executions of this sample */
   EXEC SQL DELETE FROM emp_resume WHERE empno = 'A00130';

   /* Use a single row select to get the document */
   EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
      WHERE empno = '000130' AND resume_format = 'ascii';   2
   CHECKERR ("SELECT");

   /* Use the POSSTR function to locate the start of
      sections "Department Information" & "Education" */
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
      INTO :hv_start_deptinfo; 3
   CHECKERR ("VALUES1");

   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
      INTO :hv_start_educ;
   CHECKERR ("VALUES2");
   /* Replace Department Information Section with nothing */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, 1, :hv_start_deptinfo -1)
      || SUBSTR (:hv_doc_locator1, :hv_start_educ))
      INTO :hv_doc_locator2;
   CHECKERR ("VALUES3");

   /* Move Department Information Section into the hv_new_section_buffer */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, :hv_start_deptinfo,
      :hv_start_educ -:hv_start_deptinfo)) INTO :hv_new_section_buffer;
   CHECKERR ("VALUES4");

   /* Append our new section to the end (assume it has been filled in)
      Effectively, this just moves the Department Information to the bottom
      of the resume. */
   EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer) INTO
      :hv_doc_locator3;
   CHECKERR ("VALUES5");

   /* Store this resume section in the table. This is where the LOB value
      bytes really move */
   EXEC SQL INSERT INTO emp_resume VALUES ('A00130', 'ascii',
      :hv_doc_locator3);   4
   CHECKERR ("INSERT");

   printf ("LOBEVAL completed\n");

   /* free the locators */   5
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2, : hv_doc_locator3;
   CHECKERR ("FREE LOCATOR");

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : LOBEVAL.SQC */
```

COBOL Sample: LOBEVAL.SQB

```
 Identification Division.
 Program-ID. "lobeval".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 hv-start-deptinfo pic s9(9) comp-5.
 01 hv-start-educ     pic s9(9) comp-5.
 01 hv-return-code    pic s9(9) comp-5.
 01 hv-new-section-buffer USAGE IS SQL TYPE IS CLOB(5K).
 01 hv-doc-locator1   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator2   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator3   USAGE IS SQL TYPE IS CLOB-LOCATOR.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBEVAL".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Delete any instance of "A00130" from previous executions
     EXEC SQL DELETE FROM emp_resume
              WHERE empno = 'A00130' END-EXEC.

* use a single row select to get the document
     EXEC SQL SELECT resume INTO :hv-doc-locator1  2
              FROM emp_resume
              WHERE empno = '000130'
              AND resume_format = 'ascii' END-EXEC.
     move "SELECT" to errloc.
     call "checkerr" using SQLCA errloc.

* use the POSSTR function to locate the start of sections
* "Department Information" & "Education"
     EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
              'Department Information'))
              INTO :hv-start-deptinfo END-EXEC.  3
     move "VALUES1" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
              'Education')) INTO :hv-start-educ END-EXEC.
     move "VALUES2" to errloc.
     call "checkerr" using SQLCA errloc.

* replace Department Information section with nothing
     EXEC SQL VALUES (SUBSTR(:hv-doc-locator1, 1,
              :hv-start-deptinfo - 1) ||
              SUBSTR(:hv-doc-locator1, :hv-start-educ))
              INTO :hv-doc-locator2 END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.

* move Department Information section into hv-new-section-buffer
     EXEC SQL VALUES (SUBSTR(:hv-doc-locator1,
              :hv-start-deptinfo,
              :hv-start-educ - :hv-start-deptinfo))
              INTO :hv-new-section-buffer END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

* Append the new section to the end (assume it has been filled)
* Effectively, this just moves the Dept Info to the bottom of
* the resume.
     EXEC SQL VALUES (:hv-doc-locator2 ||
              :hv-new-section-buffer)
              INTO :hv-doc-locator3 END-EXEC.
     move "VALUES5" to errloc.
     call "checkerr" using SQLCA errloc.

* Store this resume in the table.
* This is where the LOB value bytes really move.
     EXEC SQL INSERT INTO emp_resume                              4
              VALUES ('A00130', 'ascii', :hv-doc-locator3)
              END-EXEC.
     move "INSERT" to errloc.
     call "checkerr" using SQLCA errloc.

     display "LOBEVAL completed".

     EXEC SQL FREE LOCATOR :hv-doc-locator1, :hv-doc-locator2 5
              :hv-doc-locator3 END-EXEC.
     move "FREE LOCATOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.

 End-Prog.
     stop run.
```

Indicator Variables and LOB Locators

For normal host variables in an application program, when selecting a NULL value into a host variable, a negative value is assigned to the indicator variable signifying that the value is NULL. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a locator host variable itself can never be NULL, a negative indicator variable value indicates that the LOB value represented by the LOB locator is NULL. The NULL information is kept local to the client using the indicator variable value — the server does not track NULL values with valid locators.

LOB File Reference Variables

File reference variables are similar to host variables except they are used to transfer data to and from client files, and not to and from memory buffers. A file reference variable represents (rather than contains) the file, just as a LOB locator represents (rather than contains) the LOB value. Database queries, updates, and inserts may use file reference variables to store, or to retrieve, single LOB values.

For very large objects, files are natural containers. In fact, it is likely that most LOBs begin as data stored in files on the client before they are moved to the database on the server. The use of file reference variables assists in moving LOB data. Programs use file reference variables to transfer LOB data from the client file directly to the database engine. The client application does not have to write utility routines to read and write files using host variables (which have size restrictions) to carry out the movement of LOB data.

Note: The file referenced by the file reference variable must be accessible from (but not necessarily resident on) the system on which the program runs. For a stored procedure, this would be the server.

A file reference variable has a data type of BLOB, CLOB, or DBCLOB. It is used either as the source of data (input) or as the target of data (output). The file reference variable may have a relative file name or a complete path name of the file (the latter is advised). The file name length is specified within the application program. The data length portion of the file reference variable is unused during input. During output, the data length is set by the application requestor code to the length of the new data written to the file.

When using file reference variables there are different options on both input and output. You must choose an action for the file by setting the `file_option` field in the file reference variable structure. Choices for assignment to the field covering both input and output values are shown below.

Values (shown for C) and options when using input file reference variables are as follows:

- **SQL_FILE_READ** (Regular file) — This is a file that can be open, read, and closed. DB2 determines the length of the data in the file (in bytes) when opening the file. DB2 then returns the length through the `data_length` field of the file reference variable structure. (The value for COBOL is SQL-FILE-READ, and for FORTRAN is sql_file_read.)

Values and options when using output file reference variables are as follows:

- **SQL_FILE_CREATE** (New file) — This option creates a new file. Should the file already exist, an error message is returned. (The value for COBOL is SQL-FILE-CREATE, and for FORTRAN is sql_file_create.)
- **SQL_FILE_OVERWRITE** (Overwrite file) — This option creates a new file if none already exists. If the file already exists, the new data overwrites the data in the file. (The value for COBOL is SQL-FILE-OVERWRITE, and for FORTRAN is sql_file_overwrite.)
- **SQL_FILE_APPEND** (Append file) — This option has the output appended to the file, if it exists. Otherwise, it creates a new file. (The value for COBOL is SQL-FILE-APPEND, and for FORTRAN is sql_file_append.)

Notes:

1. In an Extended UNIX Code (EUC) environment, the files to which DBCLOB file reference variables point are assumed to contain valid EUC characters appropriate for storage in a graphic column, and to never contain UCS-2 characters. For more information on DBCLOB files in an EUC environment, see “Considerations for DBCLOB Files” on page 381.
2. If a LOB file reference variable is used in an OPEN statement, the file associated with the LOB file reference variable must not be deleted until the cursor is closed.

For more information on file reference variables, see the *SQL Reference*.

Example: Extracting a Document To a File

This program example shows how CLOB elements can be retrieved from a table into an external file.

How the Sample LOBFILE Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. A CLOB FILE REFERENCE host variable is declared.
2. **CLOB FILE REFERENCE host variable is set up.** The attributes of the FILE REFERENCE is set up. A file name without a fully declared path is, by default, placed in the current working directory.
3. **Select in to the CLOB FILE REFERENCE host variable.** The data from the `resume` field is selected into the filename referenced by the host variable.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C `check_error` is redefined as `CHECKERR` and is located in the `util.c` file.

COBOL `CHECKERR` is an external program named `checkerr.cbl`

See “Using GET ERROR MESSAGE in Example Programs” on page 85 for the source code for this error checking utility.

C Sample: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;  1
      SQL TYPE IS CLOB_FILE resume;
      short lobind;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBFILE\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobfile [userid passwd]\n\n");
      return 1;
   } /* endif */

   strcpy (resume.name, "RESUME.TXT");   2
   resume.name_length = strlen("RESUME.TXT");
   resume.file_options = SQL_FILE_OVERWRITE;

   EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume   3
      WHERE resume_format='ascii' AND empno='000130';

   if (lobind < 0) {
      printf ("NULL LOB indicated \n");
   } else {
      printf ("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
   } /* endif */

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : LOBFILE.SQC */
```

COBOL Sample: LOBFILE.SQB

```
 Identification Division.
 Program-ID. "lobfile".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 resume            USAGE IS SQL TYPE IS CLOB-FILE.
 01 lobind            pic s9(4) comp-5.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBFILE".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "RESUME.TXT" to resume-NAME.                             2
     move 10 to resume-NAME-LENGTH.
     move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

     EXEC SQL SELECT resume INTO :resume :lobind  3
              FROM emp_resume
              WHERE resume_format = 'ascii'
              AND empno = '000130' END-EXEC.
     if lobind less than 0 go to NULL-LOB-indicated.

     display "Resume for EMPNO 000130 is in file : RESUME.TXT".
     go to End-Main.
```

```
NULL-LOB-indicated.
    display "NULL LOB indicated".

End-Main.
    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Prog.
           stop run.
```

Example: Inserting Data Into a CLOB Column

In the path description of the following C program segment:

- `userid` represents the directory for one of your users.
- `dirname` represents a subdirectory name of “userid.”
- `filnam.1` can become the name of one of your documents that you wish to insert into the table.
- `clobtab` is the name of the table with the CLOB data type.

The following example shows how to insert data from a regular file referenced by `:hv_text_file` into a CLOB column (note that the path names used in the example are for UNIX-based systems):

```
strcpy(hv_text_file.name, "/u/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/u/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

User-Defined Functions (UDF)

A user-defined function is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. Thus, you may need to extend SQL for the following reasons:

- **Customization**.

  The function specific to your application does not exist in DB2. Whether the function is a simple transformation, a trivial calculation, or a complicated multivariate analysis, you can probably use a UDF to do the job.

- **Flexibility**.

  The DB2 built-in function does not quite permit the variations that you wish to include in your application.

- **Standardization**.

  Many of the programs at your site implement the same basic set of functions, but there are minor differences in all the implementations. Thus, you are unsure about the consistency of the results you receive. If you correctly implement these functions once, in a UDF, then all these programs can use the same implementation directly in SQL and provide consistent results.

- **Object-relational support**.

  As discussed in “User-defined Distinct Types (UDT)” on page 271, UDTs can be very useful in extending the capability and increasing the safety of DB2. UDFs act as the *methods* for UDTs, by providing behavior and encapsulating the types.

Why Use UDFs?

In writing DB2 applications, you have a choice when implementing desired actions or operations:

- As a UDF
- As a subroutine or function in your application.

Although it may seem easier to implement new operations as subroutines or functions in your application, there are good reasons why you should consider using UDFs:

- **Re-use**.

  If the new operation is something of which other users or programs at your site can take advantage, then UDFs can help to reuse it. In addition, the function can be invoked directly in SQL wherever an expression can be used by any user of the database. The database will take care of many data type promotions of the function arguments automatically, for example DECIMAL to DOUBLE, allowing your function to be applied to different, but compatible data types.

  It may seem easier to implement your new function as a normal function and then make it available to others for use in their programs, thereby avoiding the need to define the function to DB2. This requires that you inform all other interested application developers, and package the function effectively for their use. However, it ignores the interactive users like those who normally use the Command Line Processor (CLP) to access the database. CLP users cannot use your function unless it is a UDF in the database. This also applies to any other tools that use SQL (such as Visualizer), that do not get recompiled.

- **Performance**.

  Invoking the UDF directly from the database engine instead of from your application can have a considerable performance advantage, particularly when the function may be used in the qualification of data for further processing. Consider a simple scenario where you want to process some data, provided you can meet some selection criteria which can be expressed as a function `SELECTION_CRITERIA()`. Your application could issue the following select statement:

  ```
  SELECT A,B,C FROM T
  ```

  When it receives each row, it runs `SELECTION_CRITERIA` against the data to decide if it is interested in processing the data further. Here, every row of table T must be passed back to the application. But if `SELECTION_CRITERIA()` is implemented as a UDF, your application can issue the following statement:

  ```
  SELECT A,B,C FROM T WHERE SELECTION_CRITERIA(A,B) = 1
  ```

  In this case, only the rows of interest are passed across the interface between the application and the database. For large tables, or for cases where `SELECTION_CRITERIA` supplies significant filtering, the performance improvement can be very significant.

  Another case where a UDF can offer a performance benefit is when dealing with Large Objects (LOB). If you have a function whose purpose is to extract some information from a value of one of the LOB types, you can perform this extraction right on the database server and pass only the extracted value back to the application. This is more efficient than passing the entire LOB value back to the application and then performing the extraction. The performance value of packaging this function as a UDF could be enormous, depending on the particular situation. (Note that you can also extract a portion of a LOB by using a LOB locator. See “Example: Deferring the Evaluation of a LOB Expression” on page 242 for an example of a similar scenario.)

  In addition, with table functions, you can very efficiently use relational operations and the power of SQL on data that resides outside a DB2 database (including non-relational data stores). A table function takes individual scalar values of different types and meanings as its arguments, and returns a table to the SQL statement that invokes it. You can write table functions that generate only the data in which you are interested, eliminating any unwanted rows or columns. For more information on table functions, including rules on where you can use them, see the section on *Functions* in the *SQL Reference*.

- **Object Orientation**.

  You can implement the behavior of a user-defined distinct type (UDT), also called *distinct type*, using a UDF. For more information on UDTs, see “User-defined Distinct Types (UDT)” on page 271. For additional details on UDTs and the important concept of *castability* discussed herein, see the CREATE DISTINCT

TYPE statement in the *SQL Reference*. When you create a distinct type, you are automatically provided cast functions between the distinct type and its source type, and you may be provided comparison operators such as =, >, <, and so on, depending on the source type. You have to provide any additional behavior yourself. Because it is clearly desirable to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it, UDFs can be used as the implementation mechanism.

For example, suppose you have a BOAT distinct type, defined over a one megabyte BLOB. The BLOB contains the various nautical specifications, and some drawings. You may wish to compare sizes of boats, and with a distinct type defined over a BLOB source type, you do not get the comparison operations automatically generated for you. You can implement a BOAT_COMPARE function which decides if one boat is bigger than another based on a measure that you choose. These could be: displacement, length over all, metric tonnage, or another calculation based on the BOAT object. You create the BOAT_COMPARE function as follows:

```
CREATE FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
```

If your function returns 1 if the first BOAT is bigger, 2 if the second is bigger, and 0if they are equal, you could use this function in your SQL code to compare boats. Suppose you create the following tables:

```
CREATE TABLE BOATS_INVENTORY (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  OWNER          VARCHAR(40),
  DESIGN_DATE    DATE,
  SPEC           BOAT,
  ...                  )

CREATE TABLE MY_BOATS (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  DESIGN_DATE    DATE,
  ACQUIRE_DATE   DATE,
  ACQUIRE_PRICE  CANADIAN_DOLLAR,
  CURR_APPRAISL  CANADIAN_DOLLAR,
  SPEC           BOAT,
  ...                  )
```

You can execute the following SQL SELECT statement:

```
SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
  INV.OWNER, INV.DESIGN_DATE
  FROM BOATS_INVENTORY INV, MY_BOATS MY
  WHERE MY.BOAT_ID = '19GCC'
  AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
```

This simple example returns all the boats from BOATS_INVENTORY that are bigger than a particular boat in MY_BOATS. Note that the example only passes the rows of interest back to the application because the comparison occurs in the database server. In fact, it completely avoids passing any values of data type BOAT. This is a significant improvement in storage and performance as BOAT is based on a one megabyte BLOB data type.

UDF Concepts

The following is a discussion of the important concepts you need to know prior to coding UDFs:

- Full name of a function.

  The full name of a function is `<schema-name>.<function-name>`. You can use this full name anywhere you refer to a function. For example:

  ```
  SLICKO.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
  ```

  However, you may also omit the `<schema-name>.`, in which case, DB2 must determine the function to which you are referring. For example:

  ```
  BOAT_COMPARE    FOO    SUBSTR    FLOOR
  ```

- Function Path

  The concept of *function path* is central to DB2's resolution of *unqualified* references that occur when you do not use the `schema-name`. For the use of function path in DDL statements that refer to functions, see description of the corresponding statement in the *SQL Reference*. The function path is an ordered list of schema names. It provides a set of schemas for resolving unqualified function references to UDFs as well as UDTs. In cases where a function reference matches functions in more than one schema in the path, the order of the schemas in the path is used to resolve this match. The function path is established by means of the FUNCPATH option on the precompile and bind commands for static SQL. The function path is set by the SET CURRENT FUNCTION PATH statement for dynamic SQL. The function path has the following default value:

  ```
  "SYSIBM","SYSFUN","<ID>"
  ```

  This applies to both static and dynamic SQL, where `<ID>` represents the current statement authorization ID.

- Overloaded function names.

  Function names can be *overloaded*, which means that multiple functions, even in the same schema, can have the same name. Two functions cannot, however, have the same *signature*, which can be defined to be the qualified function name concatenated with the defined data types of all the function parameters in the order in which they are defined. For an example of an overloaded function, see "Example: BLOB String Search" on page 260.

- Function selection algorithm

  It is the *function selection algorithm* that takes into account the facts of overloading and function path to choose *the best fit* for every function reference, whether it is a qualified or an unqualified reference. Even references to the built-in functions and the functions (also IBM-supplied) in the SYSFUN schema are processed through the function selection algorithm.

- Types of function.

  Each user-defined function is classified as a *scalar*, *column* or *table* function. A *scalar function* returns a single value answer each time it is called. For example, the built-in function `SUBSTR()` is a scalar function. Scalar UDFs can either be external (coded in a programming language such as C), or sourced (using the implementation of an existing function).

  A *column function* receives a set of like values (a column of data) and returns a single value answer from this set of values. These are also called *aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF that is sourced on one of the built-in column functions can be defined. This is useful for distinct types. For example, if a distinct type `SHOESIZE` exists that is defined with base type INTEGER, you could define a UDF, `AVG(SHOESIZE)`, as a column function sourced on the existing built-in column function, `AVG(INTEGER)`.

  A *table function* returns a table to the SQL statement that references it. A table function can only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply the SQL language to non-DB2 data, or to capture such data and put it into a DB2 table. For example, it could dynamically convert a file consisting of non-DB2 data into a table, or it could retrieve data from the World Wide Web or an operating system and tabularize it. A table function can only be an external function.

The concept of function path, the SET CURRENT FUNCTION PATH statement, and the function selection algorithm are discussed in detail in the *SQL Reference*. The FUNCPATH precompile and bind options are discussed in detail in the *Command Reference*.

Implementing UDFs

The process of implementing a UDF requires the following steps:

1. Writing the UDF
2. Compiling the UDF
3. Linking the UDF
4. Debugging the UDF
5. Registering the UDF with DB2.

After these steps are successfully completed, your UDF is ready for use in data manipulation language (DML) or data definition language (DDL) statements such as CREATE VIEW. The steps of writing and defining UDFs are discussed in the following sections, followed by a discussion on using UDFs. For information on compiling and linking UDFs, see the *DB2 SDK Building Applications* book for your operating system. For information on debugging your UDF, see “Debugging your UDF” on page 347.

Writing UDFs

You can find the details of how you write UDFs in Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285. This includes the details on the interface between DB2 and UDFs, UDF coding considerations, UDF coding examples, and UDF debugging information. For information on the related tasks of compiling and linking your UDFs, see the *DB2 SDK Building Applications* book for your operating system.

Registering UDFs

You should register the UDF to DB2 after you have written and completely tested the actual code. Note that it is possible to define the UDF prior to actually writing it. However, to avoid any problems with running your UDF, you are encouraged to write and test it extensively before registering it. For information on testing your UDF, see “Debugging your UDF” on page 347.

Use the CREATE FUNCTION statement to define (or register) your UDF to DB2. You can find detailed explanations for this statement and its options in the *SQL Reference*.

Examples of Registering UDFs

The examples which follow illustrate a variety of typical situations where UDFs can be registered.

Note that in these examples:

- The keyword or keyword/value specifications are always shown in the same order, for consistency of presentation and ease of understanding. In actually writing one of these CREATE FUNCTION statements, after the function name and the list of parameter data types, the specifications can appear in any order.
- The specifications in the EXTERNAL NAME clause are always shown for DB2 for AIX. You may need to make changes if you run these examples on non-UNIX platforms. For example, by converting all the slash (/) characters to back slash characters (\) and adding a drive letter such as `C:`, you have examples that are valid in OS/2 or Windows environments. See the *SQL Reference* for a complete discussion of the EXTERNAL NAME clause.

Example: Exponentiation

Suppose you have written an external UDF to perform exponentiation of floating point values, and wish to register it in the MATH schema. Assume that you have DBADM authority. As you have tested the function extensively, and know that it does not represent any integrity exposure, you define it as NOT FENCED. By virtue of having DBADM authority, you possess the database authority, CREATE_NOT_FENCED, which is required to define the function as NOT FENCED.

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/common/math/exponent'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
```

In this example, the system uses the NOT NULL CALL default value. This is desirable since you want the result to be NULL if either argument is NULL. Since you do not require a scratchpad and no final call is necessary, the NO SCRATCHPAD and NO FINAL CALL default values are used. As there is no reason why EXPON cannot be parallel, the ALLOW PARALLELISM default value is used.

Example: String Search

Your associate, Willie, has written a UDF to look for the existence of a given short string, passed as an argument, within a given CLOB value, which is also passed as an argument. The UDF returns the position of the string within the CLOB if it finds the string, or zero if it does not. Because you are concerned with database integrity for this function as you suspect the UDF is not fully tested, you define the function as FENCED.

Additionally, Willie has written the function to return a FLOAT result. Suppose you know that when it is used in SQL, it should always return an INTEGER. You can create the following function:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_find_feb95"
  EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

Note that a CAST FROM clause is used to specify that the UDF body really returns a FLOAT value but you want to cast this to INTEGER before returning the value to the statement which used the UDF. As discussed in the *SQL Reference*, the INTEGER built-in function can perform this cast for you. Also, you wish to provide your own specific name for the function and later reference it in DDL (see “Example: String Search over UDT” on page 261). Because the UDF was not written to handle NULL values, you use the NOT NULL CALL default value. And because there is no scratchpad, you use the NO SCRATCHPAD and NO FINAL CALL default values. As there is no reason why FINDSTRING cannot be parallel, the ALLOW PARALLELISM default value is used.

Example: BLOB String Search

Because you want this function to work on BLOBs as well as on CLOBs, you define another FINDSTRING taking BLOB as the first parameter:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_fblob_feb95"
  EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

This example illustrates overloading of the UDF name, and shows that multiple UDFs can share the same body. Note that although a BLOB cannot be assigned to a CLOB, the same source code can be used. There is no programming problem in the above example as the programming interface for BLOB and CLOB between DB2 and UDF is the same; length followed by data. DB2 does not check if the UDF using a particular function body is in any way consistent with any other UDF using the same body.

Example: String Search over UDT

This example is a continuation of the previous example. Say you are satisfied with the FINDSTRING functions from “Example: BLOB String Search” on page 260 , but now you have defined a distinct type BOAT with source type BLOB. You also want FINDSTRING to operate on values having data type BOAT, so you create another FINDSTRING function. This function is sourced on the FINDSTRING which operates on BLOB values in “Example: BLOB String Search” on page 260 . Note the further overloading of FINDSTRING in this example:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat_mar95"
  SOURCE SPECIFIC "willie_fblob_feb95"
```

Note that this FINDSTRING function has a different signature from the FINDSTRING functions in “Example: BLOB String Search” on page 260 , so there is no problem overloading the name. You wish to provide our own specific name for possible later reference in DDL. Because you are using the SOURCE clause, you cannot use the EXTERNAL NAME clause or any of the related keywords specifying function attributes. These attributes are taken from the source function. Finally, observe that in identifying the source function you are using the specific function name explicitly provided in “Example: BLOB String Search” on page 260 . Because this is an unqualified reference, the schema in which this source function resides must be in the function path, or the reference will not be resolved.

Example: External Function with UDT Parameter

You have written another UDF to take a BOAT and examine its design attributes and generate a cost for the boat in Canadian dollars. Even though internally, the labor cost may be priced in German marks, or Japanese yen, or US dollars, this function needs to generate the cost to build the boat in the required currency, Canadian dollars. This means it has to get current exchange rate information from the `exchange_rate` file, managed outside of DB2, and the answer depends on what it finds in this file. This makes the function NOT DETERMINISTIC (or VARIANT).

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  EXTERNAL NAME '/u/marine/funcdir/costs!boatcost'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

Observe that CAST FROM and SPECIFIC are not specified, but that NOT DETERMINISTIC is specified. Here again, FENCED is chosen for safety reasons.

Example: AVG over a UDT

This example implements the AVG column function over the CANADIAN_DOLLAR distinct type. See "Example: Money" on page 272 for the definition of CANADIAN_DOLLAR. Strong typing prevents you from using the built-in AVG function on a distinct type. It turns out that the source type for CANADIAN_DOLLAR was DECIMAL, and so you implement the AVG by sourcing it on the AVG(DECIMAL) built-in function. The ability to do this depends on being able to cast from DECIMAL to CANADIAN_DOLLAR and vice versa, but since DECIMAL is the source type for CANADIAN_DOLLAR you know these casts will work.

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE "SYSIBM".AVG(DECIMAL(9,2))
```

Note that in the SOURCE clause you have qualified the function name, just in case there might be some other AVG function lurking in your function path.

Example: Counting

Your simple counting function returns a 1 the first time and increments the result by one each time it is called. This function takes no SQL arguments, and by definition it is a NOT DETERMINISTIC function since its answer varies from call to call. It uses the scratchpad to save the last value returned, and each time it is invoked it increments this value and returns it. You have rigorously tested this function, and possess DBADM authority on the database, so you will define it as NOT FENCED. (DBADM implies CREATE_NOT_FENCED.)

```
CREATE FUNCTION COUNTER ()
  RETURNS INT
  EXTERNAL NAME '/u/roberto/myfuncs/util!ctr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NOT FENCED
  SCRATCHPAD
  DISALLOW PARALLEL
```

Note that no parameter definitions are provided, just empty parentheses. The above function specifies SCRATCHPAD, and uses the default specification of NO FINAL CALL. In this case, as the size of the 100-byte scratchpad is sufficient, no storage has to be freed by means of a final call, and so NO FINAL CALL is specified. Since the COUNTER function requires that a single scratchpad be used to operate properly, DISALLOW PARALLEL is added to prevent DB2 from operating it in parallel. To see an implementation of this COUNTER function, refer to “Example: Counter” on page 330.

EXAMPLE: Counting with an OLE automation object

This example implements the previous counting example as an OLE (Object Linking and Embedding) automation object, `counter`, with an instance variable, `nbrOfInvoke`, to keep track of the number of invocations. Every time the UDF gets invoked, the `increment` method of the object increments the `nbrOfInvoke` instance variable and returns its current state. The automation object is registered in the Windows registry with the OLE programmatic identifier (progID) `bert.bcounter`.

```
CREATE FUNCTION bcounter ()
  RETURNS integer
  EXTERNAL NAME 'bert.bcounter!increment'
  LANGUAGE OLE
  PARAMETER STYLE DB2SQL
  SCRATCHPAD
  NOT DETERMINISTIC
  FENCED
  NULL CALL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

The implementation of the class `counter` is shown in “Example: Counter OLE Automation UDF in BASIC” on page 341 and in “Example: Counter OLE Automation UDF in C++” on page 342. For details of OLE support with DB2, see “Writing OLE Automation UDFs” on page 306.

EXAMPLE: Table Function Returning Document IDs

You have written a table function which returns a row consisting of a single document identifier column for each known document in your text management system which matches a given subject area (the first parameter) and contains the given string (second parameter). This UDF uses the functions of the text management system to quickly identify the documents:

```
     CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
       RETURNS TABLE (DOC_ID CHAR(16))
       EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
       LANGUAGE C
       PARAMETER STYLE DB2SQL
       NO SQL
       DETERMINISTIC
       NO EXTERNAL ACTION
       NOT FENCED
       SCRATCHPAD
       FINAL CALL
       DISALLOW PARALLEL
       CARDINALITY 20
```

Within the context of a single session it will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH, including the column name DOC_ID, and that FINAL CALL must be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as table functions cannot operate in parallel. Although the size of the output from DOCMATCH is highly variable, CARDINALITY 20 is a representative value, and is specified to help the DB2 optimizer to make good decisions.

Typically this table function would be used in a join with the table containing the document text, as follows:

```
  SELECT T.AUTHOR, T.DOCTEXT
  FROM DOCS as T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN''S LEMMA')) as F
  WHERE T.DOCID = F.DOC_ID
```

Note the special syntax (TABLE keyword) for specifying a table function in a FROM clause. In this invocation, the `docmatch()` table function returns a row containing the single column DOC_ID for each `mathematics` document referencing `Zorn's Lemma`. These DOC_ID values are joined to the master document table, retrieving the author's name and document text.

Using UDFs

Scalar and column UDFs can be invoked within an SQL statement wherever an expression is valid (there are additional rules for all column functions that limit validity). Table UDFs can only be referenced in the FROM clause of a SELECT. The *SQL Reference* discusses all these contexts in detail. The discussion and examples used in this section focus on relatively simple SELECT statement contexts, but note that their use is not restricted to these contexts.

Refer to “UDF Concepts” on page 257 for a summary of the use and importance of the *function path* and the *function selection* algorithm. You can find the details for both of these concepts in the *SQL Reference*. The resolution of any Data Manipulation Language (DML) reference to a function uses the function selection algorithm, so it is important to understand how it works.

Referring to Functions

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:

```
►──function_name──(──┬──────────────────────┬──)──────────────────────────►
                     │  ┌─,──────────────┐  │
                     └──▼─expression─────┴──┘
```

In the above, `function_name` can be either an unqualified or a qualified function name, and the arguments can number from 0 to 90, and are expressions which may contain:

- A column name, qualified or unqualified
- A constant
- A host variable
- A special register
- A parameter marker. (For information on the limitations of parameter marker use, refer to the section in the *SQL Reference* that describes the rules for parameter markers.)

The position of the arguments is important and must conform to the function definition for the semantics to be correct. Both the position of the arguments and the function definition must conform to the function body itself. DB2 does not attempt to shuffle arguments to better match a function definition, and DB2 does not understand the semantics of the individual function parameters.

Use of column names in UDF argument expressions requires that the table references which contain the column have proper scope. For table functions referenced in a join, this means that for any argument which involves columns from another table or table function, that other table or table function must appear before the table function containing the reference, in the FROM clause. For a complete discussion of the rules for using columns in the arguments of table functions, see the chapter on Queries in the *SQL Reference*.

Examples of Function Invocations

Some valid examples of function invocations are:

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT(SELECT SUM(length*length)
     FROM triangles
     WHERE id= 'J522'
     AND legtype <> 'HYP')
```

Note that if any of the above functions are table functions, the syntax to reference them is slightly different than presented above. For example, if PABLO.BLOOP is a table function, to properly reference it, use:

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

Using Parameter Markers in Functions

An important restriction involves parameter markers; you cannot simply code the following:

```
BLOOP(?)
```

As the function selection logic does not know what data type the argument may turn out to be, it cannot resolve the reference. You can use the CAST specification to provide a type for the parameter marker, for example INTEGER, and then the function selection logic can proceed:

```
BLOOP(CAST(? AS INTEGER))
```

Using Qualified Function Reference

If you use a qualified function reference, you restrict DB2’s search for a matching function to that schema. For example, you have the following statement:

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

Only the BLOOP functions in schema PABLO are considered. It does not matter that user SERGE has defined a BLOOP function, or whether or not there is a built-in BLOOP function. Now suppose that user PABLO has defined two BLOOP functions in his schema:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

BLOOP is thus overloaded within the PABLO schema, and the function selection algorithm would choose the best BLOOP, depending on the data type of the argument, column1. In this case, both of the PABLO.BLOOPs take numeric arguments, and if column1 is not one of the numeric types, the statement will fail. On the other hand if column1 is either SMALLINT or INTEGER, function selection will resolve to the first BLOOP, while if column1 is DECIMAL or DOUBLE, the second BLOOP will be chosen.

Several points about this example:

1. It illustrates argument promotion. The first BLOOP is defined with an INTEGER parameter, yet you can pass it a SMALLINT argument. The function selection algorithm supports promotions among the built-in data types (for details, see the *SQL Reference*) and DB2 performs the appropriate data value conversions.
2. If for some reason you want to invoke the second BLOOP with a SMALLINT or INTEGER argument, you have to take an explicit action in your statement as follows:

   ```
   SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
   ```

3. Alternatively, if you want to invoke the first BLOOP with a DECIMAL or DOUBLE argument, you have your choice of explicit actions, depending on your exact intent:

   ```
   SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(CEILING(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(INTEGER(ROUND(COLUMN1,0))) FROM T
   ```

   You should investigate these other functions in the *SQL Reference*. The INTEGER function is a built-in function in the SYSIBM schema. The FLOOR, CEILING, and ROUND functions are UDFs shipped with DB2, which you can find in the SYSFUN schema along with many other useful functions.

Using Unqualified Function Reference

If, instead of a qualified function reference, you use an unqualified function reference, DB2's search for a matching function normally uses the function path to qualify the reference. In the case of the DROP FUNCTION or COMMENT ON FUNCTION functions, the reference is qualified using the current authorization ID, if they are unqualified. Thus, *it is important that you know what your function path is, and what, if any, conflicting functions exist in the schemas of your current function path.* For example, suppose you are PABLO and your static SQL statement is as follows, where COLUMN1 is data type INTEGER:

```
SELECT BLOOP(COLUMN1) FROM T
```

You have created the two BLOOP functions cited in “Using Qualified Function Reference” on page 266, and you want and expect one of them to be chosen. If the following default function path is used, the first BLOOP is chosen (since column1 is INTEGER), if there is no conflicting BLOOP in SYSIBM or SYSFUN:

```
"SYSIBM","SYSFUN","PABLO"
```

However, suppose you have forgotten that you are using a script for precompiling and binding which you previously wrote for another purpose. In this script, you explicitly coded your FUNCPATH parameter to specify the following function path for another reason that does not apply to your current work:

```
"KATHY","SYSIBM","SYSFUN","PABLO"
```

If Kathy has written a BLOOP function for her own purposes, the function selection could very well resolve to Kathy’s function, and your statement would execute without error. You are not notified because DB2 assumes that you know what you are doing. It becomes your responsibility to identify the incorrect output from your statement and make the required correction.

Summary of Function References

For both qualified and unqualified function references, the function selection algorithm looks at all the applicable functions, both built-in and user-defined, that have:

- The given name
- The same number of defined parameters as arguments in the function reference
- Each parameter identical to or promotable from the type of the corresponding argument.

(*Applicable functions* means *functions in the named schema* for a qualified reference, or *functions in the schemas of the function path* for an unqualified reference.) The algorithm looks for an exact match, or failing that, a best match among these functions. The current function path is used, in the case of an unqualified reference only, as the deciding factor if two identically good matches are found in different schemas. The details of the algorithm can be found in the *SQL Reference*.

An interesting feature, illustrated by the examples at the end of “Using Qualified Function Reference” on page 266, is *the fact that function references can be nested*, even references to the same function. This is generally true for built-in functions as well as UDFs; however, there are some limitations when column functions are involved.

Refining an earlier example:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

Now consider the following DML statement:

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

If column1 is a DECIMAL or DOUBLE column, the inner BLOOP reference resolves to the second BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP resolves to the first BLOOP.

Alternatively, if column1 is a SMALLINT or INTEGER column, the inner bloop reference resolves to the first BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP also resolves to the first BLOOP. In this case, you are seeing nested references to the same function.

A few additional points important for function references are:

- By defining a function with the name of one of the SQL operators, you can actually invoke a UDF using *infix notation*. For example, suppose you can attach some meaning to the "+" operator for values which have distinct type BOAT. You can define the following UDF:

  ```
  CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
  ```

  Then you can write the following valid SQL statement:

  ```
  SELECT BOAT_COL1 + BOAT_COL2
  FROM BIG_BOATS
  WHERE BOAT_OWNER = 'Nelson Mattos'
  ```

  But you can also write the equally valid statement:

  ```
  SELECT "+"(BOAT_COL1, BOAT_COL2)
  FROM BIG_BOATS
  WHERE BOAT_OWNER = 'Nelson Mattos'
  ```

  Note that you are not permitted to overload the built-in conditional operators such as >, =, LIKE, IN, and so on, in this way. See “Example: Integer Divide Operator” on page 322 for an example of a UDF which overloads the divide (/) operator.

- The function selection algorithm does not consider the context of the reference in resolving to a particular function. Look at these BLOOP functions, modified a bit from before:

  ```
  CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
  CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
  ```

  Now suppose you write the following SELECT statement:

  ```
  SELECT 'ABCDEFG' CONCAT BLOOP(SMALLINT_COL) FROM T
  ```

  Because the best match, resolved using the SMALLINT argument, is the first BLOOP defined above, the second operand of the CONCAT resolves to data type INTEGER. The statement fails because CONCAT demands string arguments. If the first BLOOP was not present, the other BLOOP would be chosen and the statement execution would be successful.

  Another type of contextual inconsistency that causes a statement to fail is if a given function reference resolves to a table function in a context that requires a scalar or column function. The reverse could also occur. A reference could resolve to a scalar or column function when a table function is necessary.

- UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. DB2 will materialize the entire LOB value in storage before invoking such a function, even if the source of the value is a *LOB locator* host variable. For example, consider the following fragment of a C language application:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS CLOB(150K) clob150K ;        /* LOB host var */
    SQL TYPE IS CLOB_LOCATOR clob_locator1;  /* LOB locator host var */
    char                  string[40];       /* string host var */
  EXEC SQL END DECLARE SECTION;
  ```

Either host variable :clob150K or :clob_locator1 is valid as an argument for a function whose corresponding parameter is defined as CLOB(500K). Thus, referring to the FINDSTRING defined in “Example: String Search” on page 260, both of the following are valid in the program:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- UDF parameters or results which have one of the LOB types can be created with the AS LOCATOR modifier. In this case, the entire LOB value is not materialized prior to invocation. Instead, a LOB LOCATOR is passed to the UDF, which can then use the special UDF APIs to manipulate the actual bytes of the LOB value (see “Using LOB Locators as UDF Parameters or Results” on page 315 for details).

  You can also use this capability on UDF parameters or results which have a distinct type that is based on a LOB. This capability is limited to UDFs defined as not-fenced. Note that the argument to such a function can be any LOB value of the defined type; it does not have to be a host variable defined as one of the LOCATOR types. The use of host variable locators as arguments is completely orthogonal to the use of AS LOCATOR in UDF parameters and result definitions.

- UDFs can be defined with distinct types as parameters or as the result. (Earlier examples have illustrated this.) DB2 will pass the value to the UDF in the format of the source data type of the distinct type.

  Distinct type values which originate in a host variable and which are used as arguments to a UDF which has its corresponding parameter defined as a distinct type, **must be explicitly cast to the distinct type by the user**. There is no host language type for distinct types. DB2’s strong typing necessitates this. Otherwise your results may be ambiguous. So, consider the BOAT distinct type which is defined over a BLOB, and consider the BOAT_COST UDF from “Example: External Function with UDT Parameter” on page 261, which takes an object of type BOAT as its argument. In the following fragment of a C language application, the host variable :ship holds the BLOB value that is to passed to the BOAT_COST function:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS BLOB(150K) ship;
EXEC SQL END DECLARE SECTION;
```

  Both of the following statements correctly resolve to the BOAT_COST function, because both cast the :ship host variable to type BOAT:

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

  If there are multiple BOAT distinct types in the database, or BOAT UDFs in other schema, you must exercise care with your function path. Otherwise your results may be ambiguous.

User-defined Distinct Types (UDT)

Enabling you to define new data types to DB2 gives you considerable power and extends your capabilities with the built-in data types. You are no longer restricted to using system-supplied built-in data types to model your businesses and capture the semantics of your data. A user-defined distinct type is the DB2 mechanism you can use for this purpose.

Why Use UDTs?

There are several benefits associated with UDTs:

1. **Extensibility**.

   By defining new types, you can indefinitely increase the set of types provided by DB2 to support your applications.

2. **Flexibility**.

   You can specify any semantics and behavior for your new type by using user-defined functions (UDFs) to augment the diversity of the types available in the system.

3. **Consistent behavior**.

   Strong typing insures that your UDTs will behave appropriately. It guarantees that only functions defined on your UDT can be applied to instances of the UDT.

4. **Encapsulation**.

   The behavior of your UDTs is restricted by the functions and operators that can be applied on them. This provides flexibility in the implementation since running applications do not depend on the internal representation that you chose for your type.

5. **Extensible behavior**.

   The definition of user-defined functions on types can augment the functionality provided to manipulate your UDT at any time. (See “User-Defined Functions (UDF)” on page 254)

6. **Performance**.

   Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code used to implement built-in functions, comparison operators, indexes, etc. for built-in data types.

7. **Foundation for object-oriented extensions**.

   UDTs are the foundation for most object-oriented features. They represent the most important step towards object-oriented extensions.

Defining a UDT

UDTs, like other objects such as tables, indexes, and UDFs, need to be defined with a CREATE statement.

Use the CREATE DISTINCT TYPE statement to define your new UDT. Detailed explanations for the statement syntax and all its options are found in the *SQL Reference*.

For the CREATE DISTINCT TYPE statement, note that:

1. The name of the new UDT can be a qualified or an unqualified name. If it is qualified by a schema different from the authorization ID of the statement, you must have DBADM authority on the database.
2. The source type of the UDT is the type used by DB2 to internally represent the UDT. For this reason, it must be a built-in data type. Previously defined UDTs cannot be used as source types of other UDTs.
3. The WITH COMPARISONS clause is used to tell DB2 that functions to support the comparison operations on instances of the UDT should be generated by DB2. This clause is required if comparison operations are supported on the source type (for example, INTEGER and DATE) and is prohibited if comparison operations are not supported (for example, LONG VARCHAR and BLOB).

Note: As part of a UDT definition, DB2 always generates cast functions to:

- Cast from the UDT to the source type, using the standard name of the source type. For example, if you create a distinct type based on FLOAT, the cast function called DOUBLE is created.
- Cast from the source type to the UDT. See the *SQL Reference* for a discussion of when additional casts to the UDTs are generated.

These functions are important for the manipulation of UDTs in queries.

Resolving Unqualified UDTs

The function path is used to resolve any references to an unqualified type name or function, except if the type name or function is

- Created
- Dropped
- Commented on.

For information on how unqualified function references are resolved, see “Using Qualified Function Reference” on page 266.

Examples of Using CREATE DISTINCT TYPE

The following are examples of using CREATE DISTINCT TYPE:

Example: Money

Suppose you are writing applications that need to handle different currencies and wish to ensure that DB2 does not allow these currencies to be compared or manipulated directly with one another in queries. Remember that conversions are necessary whenever you want to compare values of different currencies. So you define as many UDTs as you need; one for each currency that you may need to represent:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE GERMAN_MARK AS DECIMAL (9,2) WITH COMPARISONS
```

Note that you have to specify WITH COMPARISONS since comparison operators are supported on DECIMAL (9,2).

Example: Resume

Suppose you would like to keep the form filled by applicants to your company in a DB2 table and you are going to use functions to extract the information from these forms. Because these functions cannot be applied to regular character strings (because they are certainly not able to find the information they are supposed to return), you define a UDT to represent the filled forms:

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

Because DB2 does not support comparisons on CLOBs, you do not specify the clause WITH COMPARISONS. You have specified a schema name different from your authorization ID since you have DBADM authority, and you would like to keep all UDTs and UDFs dealing with applicant forms in the same schema.

Defining Tables with UDTs

After you have defined several UDTs, you can start defining tables with columns whose types are UDTs. Following are examples using CREATE TABLE:

Example: Sales

Suppose you want to define tables to keep your company's sales in different countries as follows:

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         GERMAN_MARK)
```

The UDTs in the above examples are created using the same CREATE DISTINCT TYPE statements in “Example: Money” on page 272. Note that the above examples use check constraints. For information on check constraints see the *SQL Reference*.

Example: Application Forms

Suppose you need to define a table where you keep the forms filled out by applicants as follows:

```
CREATE TABLE APPLICATIONS
  (ID               SYSIBM.INTEGER,
   NAME             VARCHAR (30),
   APPLICATION_DATE SYSIBM.DATE,
   FORM             PERSONAL.APPLICATION_FORM)
```

You have fully qualified the UDT name because its qualifier is not the same as your authorization ID and you have not changed the default function path. Remember that whenever type and function names are not fully qualified, DB2 searches through the schemas listed in the current function path and looks for a type or function name matching the given unqualified name. Because `SYSIBM` is always considered (if it has been omitted) in the current function path, you can omit the qualification of built-in data types. For example, you can execute `SET CURRENT FUNCTION PATH = cheryl` and the value of the current function path special register will be `"CHERYL"`, and does not include `"SYSIBM"`. Now, if CHERYL.INTEGER type is not defined, the statement `CREATE TABLE FOO(COL1 INTEGER)` still succeeds because SYSIBM is always considered as COL1 is of type SYSIBM.INTEGER.

You are, however, allowed to fully qualify the built-in data types if you wish to do so. Details about the use of the current function path are discussed in the *SQL Reference*.

Manipulating UDTs

One of the most important concepts associated with UDTs is *strong typing*. Strong typing guarantees that only functions and operators defined on the UDT can be applied to its instances.

Strong typing is important to ensure that the instances of your UDTs are correct. For example, if you have defined a function to convert US dollars to Canadian dollars according to the current exchange rate, you do not want this same function to be used to convert German marks to Canadian dollars because it will certainly return the wrong amount.

As a consequence of strong typing, DB2 does not allow you to write queries that compare, for example, UDT instances with instances of the UDT source type. For the same reason, DB2 will not let you apply functions defined on other types to UDTs. If you want to compare instances of UDTs with instances of another type, you have to cast the instances of one or the other type. In the same sense, you have to cast the UDT instance to the type of the parameter of a function that is not defined on a UDT if you want to apply this function to a UDT instance.

Examples of Manipulating UDTs

The following are examples of manipulating UDTs:

- Example: Comparisons Between UDTs and Constants
- Example: Casting Between UDTs
- Example: Comparisons Involving UDTs
- Example: Sourced UDFs Involving UDTs
- Example: Assignments Involving UDTs
- Example: Assignments in Dynamic SQL
- Example: Assignments Involving Different UDTs
- Example: Use of UDTs in UNION

Example: Comparisons Between UDTs and Constants

Suppose you want to know which products sold more than US $100 000.00 in the US in the month of July, 1992 (7/92).

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1992
```

Because you cannot compare US dollars with instances of the source type of US dollars (that is, DECIMAL) directly, you have used the cast function provided by DB2 to cast from DECIMAL to US dollars. You can also use the other cast function provided by DB2 (that is, the one to cast from US dollars to DECIMAL) and cast the column total to DECIMAL. Either way you decide to cast, from or to the UDT, you can use the cast specification notation to perform the casting, or the functional notation. That is, you could have written the above query as:

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1992
```

Example: Casting Between UDTs

Suppose you want to define a UDF that converts Canadian dollars to U.S. dollars. Suppose you can obtain the current exchange rate from a file managed outside of DB2. You would then define a UDF that obtains a value in Canadian dollars, accesses the exchange rate file, and returns the corresponding amount in U.S. dollars.

At first glance, such a UDF may appear easy to write. However, C does not support DECIMAL values. The UDTs representing different currencies have been defined as DECIMAL. Your UDF will need to receive and return DOUBLE values, since this is the only data type provided by C that allows the representation of a DECIMAL value without losing the decimal precision. Thus, your UDF should be defined as follows:

```
     CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
       EXTERNAL NAME '/u/finance/funcdir/currencies!cdn2us'
       LANGUAGE C
       PARAMETER STYLE DB2SQL
       NO SQL
       NOT DETERMINISTIC
       NO EXTERNAL ACTION
       FENCED
```

The exchange rate between Canadian and U.S. dollars may change between two invocations of the UDF, so you declare it as NOT DETERMINISTIC.

The question now is, how do you pass Canadian dollars to this UDF and get U.S. dollars from it? The Canadian dollars must be cast to DECIMAL values. The DECIMAL values must be cast to DOUBLE. You also need to have the returned DOUBLE value cast to DECIMAL and the DECIMAL value cast to U.S. dollars.

Such casts are performed automatically by DB2 anytime you define sourced UDFs, whose parameter and return type do not exactly match the parameter and return type of the source function. Therefore, you need to define two sourced UDFs. The first brings the DOUBLE values to a DECIMAL representation. The second brings the DECIMAL values to the UDT. That is, you define the following:

```
     CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
       SOURCE CDN_TO_US_DOUBLE (DOUBLE)

     CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
       SOURCE CDN_TO_US_DEC (DECIMAL())
```

Note that an invocation of the US_DOLLAR function as in US_DOLLAR(C1), where C1 is a column whose type is Canadian dollars, has the same effect as invoking:

```
    US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1)))))
```

That is, C1 (in Canadian dollars) is cast to decimal which in turn is cast to a double value that is passed to the CDN_TO_US_DOUBLE function. This function accesses the exchange rate file and returns a double value (representing the amount in U.S. dollars) that is cast to decimal, and then to U.S. dollars.

A function to convert German Marks to U.S. dollars would be similar to the example above:

```
CREATE FUNCTION GERMAN_TO_US_DOUBL(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!german2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED

CREATE FUNCTION GERMAN_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE GERMAN_TO_US_DOUBL (DOUBLE)

CREATE FUNCTION US_DOLLAR(GERMAN_MARK) RETURNS US_DOLLAR
  SOURCE GERMAN_TO_US_DEC (DECIMAL())
```

Example: Comparisons Involving UDTs

Suppose you want to know which products sold more in the US than in Canada and Germany for the month of March, 1989 (3/89):

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
  WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
  AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
  AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
  AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
  AND US.MONTH = 3
  AND US.YEAR  = 1989
  AND CDN.MONTH = 3
  AND CDN.YEAR  = 1989
  AND GERMAN.MONTH = 3
  AND GERMAN.YEAR  = 1989
```

Because you cannot directly compare US dollars with Canadian dollars or German Marks, you use the UDF to cast the amount in Canadian dollars to US dollars, and the UDF to cast the amount in German Marks to US dollars. You cannot cast them all to DECIMAL and compare the converted DECIMAL values because the amounts are not monetarily comparable. That is, the amounts are not in the same currency.

Example: Sourced UDFs Involving UDTs

Suppose you have defined a sourced UDF on the built-in SUM function to support SUM on German Marks:

```
CREATE FUNCTION SUM (GERMAN_MARKS)
  RETURNS GERMAN_MARKS
  SOURCE SYSIBM.SUM (DECIMAL())
```

You want to know the total of sales in Germany for each product in the year of 1994. You would like to obtain the total sales in US dollars:

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

You could not write SUM (us_dollar (total)), unless you had defined a SUM function on US dollar in a manner similar to the above.

Example: Assignments Involving UDTs

Suppose you want to store the form filled by a new applicant into the database. You have defined a host variable containing the character string value used to represent the filled form:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

INSERT INTO APPLICATIONS
  VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

You do not explicitly invoke the cast function to convert the character string to the UDT personal.application_form because DB2 lets you assign instances of the source type of a UDT to targets having that UDT.

Example: Assignments in Dynamic SQL

If you want to use the same statement given in “Example: Assignments Involving UDTs” in dynamic SQL, you can use parameter markers as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
  long id;
  char name[30];
  SQL TYPE IS CLOB(32K) form;
  char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command,"INSERT INTO APPLICATIONS VALUES");
strcat(command,"(?, ?, CURRENT DATE, CAST (? AS CLOB(32K)))");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

You made use of DB2’s cast specification to tell DB2 that the type of the parameter marker is CLOB(32K), a type that is assignable to the UDT column. Remember that you cannot declare a host variable of a UDT type, since host languages do not support UDTs. Therefore, you cannot specify that the type of a parameter marker is a UDT.

Example: Assignments Involving Different UDTs

Suppose you have defined two sourced UDFs on the built-in SUM function to support SUM on US and Canadian dollars, similar to the UDF sourced on German Marks in "Example: Sourced UDFs Involving UDTs" on page 277:

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())
```

Now suppose your supervisor requests that you maintain the annual total sales in US dollars of each product and in each country, in separate tables:

```
CREATE TABLE US_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE GERMAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

INSERT INTO US_SALES_94
  SELECT PRODUCT_ITEM, SUM (TOTAL)
  FROM US_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM

INSERT INTO CANADIAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM CANADIAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

You explicitly cast the amounts in Canadian dollars and German Marks to US dollars since different UDTs are not directly assignable to each other. You cannot use the cast specification syntax because UDTs can only be cast to their own source type.

Example: Use of UDTs in UNION

Suppose you would like to provide your American users with a view containing all the sales of every product of your company:

```
CREATE VIEW ALL_SALES AS
  SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
  FROM US_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM CANADIAN_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM GERMAN_SALES
```

You cast Canadian dollars to US dollars and German Marks to US dollars because UDTs are union compatible only with the same UDT. Note that you have to use the functional notation to cast between UDTs since the cast specification only lets you cast between UDTs and their source types.

Synergy Between UDTs, UDFs, and LOBs

In previous sections, you learned how to define and use the individual DB2 object extensions (UDTs, UDFs, and LOBs). However, as you will see in this section, there is a lot of synergy between these three object extensions.

Combining UDTs, UDFs, and LOBs

According to the concept of object-orientation, similar objects in the application domain are grouped into related types. Each of these types have a name, an internal representation, and behavior. By using UDTs, you can tell DB2 the name of your new type and how it is internally represented. A LOB is one of the possible internal representations for your new type and is the most suitable representation for large, complex structures. By using UDFs, you can define the behavior of the new type. Consequently, there is an important synergy between UDTs, UDFs, and LOBs. An application type with a complex data structure and behavior is modeled as a UDT that is internally represented as a LOB, with its behavior implemented by UDFs. As described in Chapter 8, “Using the Active DBMS Capabilities” on page 349, the rules governing the semantic integrity of your application type will be represented as constraints and triggers. To have better control and organization of your related UDTs and UDFs, you should keep them in the same schema.

Examples of Complex Applications

The following examples show how you can use UDTs, UDFs, and LOBs together in complex applications:

Example: Defining the UDT and UDFs
Example: Exploiting LOB Function to Populate the Database
Example: Exploiting UDFs to Query Instances of UDTs
Example: Exploiting LOB Locators to Manipulate UDT Instances

Example: Defining the UDT and UDFs

Suppose you would like to keep the electronic mail (e-mail) sent to your company in DB2 tables. Ignoring any issues of privacy, you are planning to write queries over such e-mail to find out their subject, how often your e-mail service is being used to receive customer orders, and so on. Because e-mail can be quite large, and because it has a complex internal structure (a sender, a receiver, the subject, date, and the e-mail content), you decide to represent the e-mail by means of a UDT whose source type is a large object. You define a set of UDFs on your e-mail type, such as functions to extract the subject of the e-mail, the sender, the date, and so on. You also define functions that can perform searches on the content of the e-mail. You do the above using the following CREATE statements:

```
CREATE DISTINCT TYPE E_MAIL AS BLOB (1M)

CREATE FUNCTION SUBJECT (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME '/u/mail/funcdir/e_mail!subject'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION SENDER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME '/u/mail/funcdir/e_mail!sender'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION RECEIVER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME '/u/mail/funcdir/e_mail!receiver'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION SENDING_DATE (E_MAIL)
  RETURNS DATE CAST FROM VARCHAR(10)
  EXTERNAL NAME '/u/mail/funcdir/e_mail!sending_date'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION CONTENTS (E_MAIL)
  RETURNS BLOB (1M)
  EXTERNAL NAME '/u/mail/funcdir/e_mail!contents'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION CONTAINS (E_MAIL, VARCHAR (200))
  RETURNS INTEGER
  EXTERNAL NAME '/u/mail/funcdir/e_mail!contains'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE TABLE ELECTRONIC_MAIL
  (ARRIVAL_TIMESTAMP  TIMESTAMP,
   MESSAGE E_MAIL)
```

Example: Exploiting LOB Function to Populate the Database

Suppose you populate your table by transferring your e-mail that is maintained in files into DB2. You would execute the following INSERT statement multiple times with different values of the HV_EMAIL_FILE until you have stored all your e_mail into DB2:

```
EXEC SQL BEGIN DECLARE SECTION
  SQL TYPE IS BLOB_FILE HV_EMAIL_FILE;

EXEC SQL END DECLARE SECTION
  strcpy (HV_EMAIL_FILE.NAME, "/u/mail/email/mbox");
  HV_EMAIL_FILE.NAME_LENGTH = strlen(HV_EMAIL_FILE.NAME);
  HV_EMAIL_FILE.FILE_OPTIONS = 2;

EXEC SQL INSERT INTO ELECTRONIC_MAIL
  VALUES (CURRENT TIMESTAMP, :hv_email_file);
```

Because all the function provided by DB2 LOB support is applicable to UDTs whose source type are LOBs, you have used LOB file reference variables to assign the contents of the file into the UDT column. You have not used the cast function to convert values of BLOB type into your e-mail type because DB2 let you assign values of the source type of a distinct type to targets to the distinct type.

Example: Exploiting UDFs to Query Instances of UDTs

Suppose you need to know how much e-mail was sent by a specific customer regarding customer orders and you have the e-mail address of your customers in the customers table.

```
SELECT COUNT (*)
  FROM ELECTRONIC_MAIL AS EMAIL, CUSTOMERS
  WHERE SUBJECT (EMAIL.MESSAGE) = 'customer order'
  AND CUSTOMERS.EMAIL_ADDRESS = SENDER (EMAIL.MESSAGE)
  AND CUSTOMERS.NAME = 'Customer X'
```

You have used the UDFs defined on the UDT in this SQL query since they are the only means to manipulate the UDT. In this sense, your UDT e-mail is completely encapsulated. That is, its internal representation and structure are hidden and can only be manipulated by the defined UDFs. These UDFs know how to interpret the data without the need to expose its representation.

Suppose you need to know the details of all the e-mail your company received in 1994 which had to do with the performance of your products in the marketplace.

```
SELECT SENDER (MESSAGE), SENDING_DATE (MESSAGE), SUBJECT (MESSAGE)
  FROM ELECTRONIC_MAIL       WHERE CONTAINS (MESSAGE,
  '"performance" AND "products" AND "marketplace"') = 1
```

You have used the `contains` UDF which is capable of analyzing the contents of the message searching for relevant keywords or synonyms.

Example: Exploiting LOB Locators to Manipulate UDT Instances

Suppose you would like to obtain information about a specific e-mail without having to transfer the entire e-mail into a host variable in your application program. (Remember that an e-mail can be quite large.) Since your UDT is defined on a LOB, you can use LOB locators for that purpose:

```
EXEC SQL BEGIN DECLARE SECTION
  long hv_len;
  char hv_subject[200];
  char hv_sender[200];
  char hv_buf[4096];
  char hv_current_time[26];
  SQL TYPE IS BLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT MESSAGE
  INTO :hv_email_locator
  FROM ELECTRONIC MAIL
  WHERE ARRIVAL_TIMESTAMP = :hv_current_time;

EXEC SQL VALUES (SUBJECT (E_MAIL(:hv_email_locator))
  INTO :hv_subject;
.... code that checks if the subject of the e_mail is relevant ....
.... if the e_mail is relevant, then..............................

EXEC SQL VALUES (SENDER (CAST (:hv_email_locator AS E_MAIL)))
  INTO :hv_sender;
```

Because your host variable is of type BLOB locator (the source type of the UDT), you have explicitly converted the BLOB locator to your UDT, whenever it was used as an argument of a UDF defined on the UDT.

Chapter 7. Writing User-Defined Functions (UDFs)

This chapter discusses the details of how you write your own UDF, the issues you need to consider, and the steps required to make your UDF operational. Specifically, the following major topics are discussed:

- Interface between DB2 and a UDF
- Writing OLE Automation UDFs
- Scratchpad Considerations
- Table Function Considerations
- Scenarios for Using LOB Locators
- Other Coding Considerations
- Examples of UDF Code
- Debugging your UDF

After a preliminary discussion on the interface between DB2 and a UDF, the remaining discussion concerns how you implement UDFs. The information on writing the UDF emphasizes the presence or absence of a scratchpad as one of the primary considerations.

Some general considerations in using this section are:

- A lot of important material on defining and using UDFs is presented in "User-Defined Functions (UDF)" on page 254, and is not repeated here. This discussion concentrates on how you implement a UDF.
- In addition to successfully running the CREATE FUNCTION statement, you must perform the following steps in order to implement an *external UDF*:
  - Writing the UDF in either C, C++ or Java
  - Compiling the UDF
  - Linking the UDF
  - Testing and Debugging the UDF

  You can find information on compiling and linking UDFs in the *DB2 SDK Building Applications* book for your platform.
- You can invoke your UDF using OLE (Object Linking and Embedding) as described in this chapter.

Note that a *sourced UDF*, which is different from an external UDF, does not require an implementation in the form of a separate piece of code. Such a UDF uses the same implementation as its source function, along with many of its other attributes.

Interface between DB2 and a UDF

This section discusses some of the details of the interface between DB2 and a UDF, and discusses the `sqludf.h` include file which makes the interface manageable. This include file only applies to C and C++ UDFs. For information on coding UDFs in Java, see "Coding a Java UDF" on page 514.

The Arguments Passed from DB2 to a UDF

In addition to the SQL arguments and those values passed in the DML reference to the function, DB2 passes additional arguments to the external UDF. For C and C++, all of these arguments are passed in the order shown in Figure 24. Java UDFs take only the *SQL-argument* and *SQL-result* arguments, leaving the other information available to extra function calls. Java UDFs have the same restrictions on the resulting *SQL-state* and *diagnostic-message* arguments documented below. For information on coding UDFs in Java, see “Coding a Java UDF” on page 514.

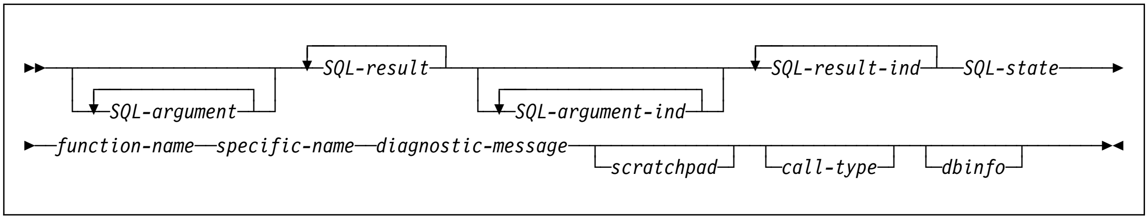


Figure 24. Passing Arguments to a UDF

Note: Each of the above arguments passed to the external function is a pointer to the value, and not the actual value.

The arguments are described as follows:

SQL-argument
This argument is set by DB2 before calling the UDF. This value repeats *n* times, where *n* is the number of arguments specified in the function reference. The value of each of these arguments is taken from the expression specified in the function invocation. It is expressed in the data type of the defined parameter. How these data types map to C language constructs is described in “How the SQL Data Types are Passed to a UDF” on page 297.

SQL-result
This argument is set by the UDF before returning to DB2. For scalar functions there is exactly one SQL-result. For table functions there is one SQL-result for each result column of the function defined in the RETURNS TABLE clause of the CREATE FUNCTION statement. They correspond to the position of the columns defined in the RETURNS TABLE clause. That is, the first SQL-result argument corresponds to the first column defined in the RETURNS TABLE clause, and so on.

For both scalar functions and table functions, DB2 allocates the buffer and passes its address to the UDF. The UDF puts each result value into the buffer. Enough buffer space is allocated by DB2 to contain the value expressed in the data type. For scalar functions, this data type is defined in the CAST FROM clause,if it is present, or in the RETURNS clause, if no CAST FROM clause is present. For table functions, the data types are defined in the RETURNS TABLE(...) clause. For information on how these types map to C language constructs, see “How the SQL Data Types are Passed to a UDF” on page 297.

Note that for table functions, DB2 defines a performance optimization where every defined column does not have to be returned to DB2. Your UDF returns only the columns required by the statement referencing the table function. For example, consider a CREATE FUNCTION statement for a table function defined with 100 result columns. If a given statement referencing the function is only interested in two of them, this optimization enables the UDF to only return those two columns for each row. See the *dbinfo* argument below for more information on this optimization.

For each value returned, (that is, a single value for a scalar function, and in general, multiple values for a table function), the UDF code should not change more bytes than is required for the data type and length of the result. DB2 will attempt to determine if the UDF body has written beyond the end of the result buffer by a few bytes, returning SQLCODE -450 (SQLSTATE 39501). However, a major overwrite by the UDF may not be detected by DB2 can cause unpredictable results or an abnormal termination,

SQL-argument-ind

This argument is set by DB2 before calling the UDF. It can be used by the UDF to determine if the corresponding *SQL-argument* is null or not. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* (described above). It contains one of the following values:

0	The argument is present and not null.
-1	The argument is present and its value is null.

If the function is defined with NOT NULL CALL, the UDF body does not need to check for a null value. However, if it is defined with NULL CALL, any argument can be NULL and the UDF should check it.

The indicator takes the form of a SMALLINT value, and this can be defined in your UDF as described in "How the SQL Data Types are Passed to a UDF" on page 297.

SQL-result-ind

This argument is set by the UDF before returning to DB2. There is one of these for each *SQL-result* argument, and they correspond positionally.

This argument is used by the UDF to signal if the particular result value is null:

0 or positive	The result is not null
negative	The result is the null value[5]

[5] DB2 treats the function result as null (-2) if the following is true:

- The database configuration parameter DFT_SQLMATHWARN is 'YES'
- One of the input arguments is a null because of an arithmetic error

Even if the function is defined with NOT NULL CALL, the UDF body must set the indicator of the result.

The indicator takes the form of a SMALLINT value, and this can be defined in your UDF as described in “How the SQL Data Types are Passed to a UDF” on page 297.

If the optimization using the *dbinfo* argument column list is being utilized, then only the indicators corresponding to the required columns need be set.

SQL-state

This argument is set by the UDF before returning to DB2. It takes the form of a CHAR(5) value. Ensure that the argument definition in the UDF is appropriate for a CHAR(5) as described in “How the SQL Data Types are Passed to a UDF” on page 297, and can be used by the UDF to signal warning or error conditions. It contains the value `'00000'`, when the function is called. The UDF can set the value to the following:

00000
The function code did not detect any warning or error situations.

01H*xx*
The function code detected a warning situation. This results in a SQL warning, SQLCODE +462 (SQLSTATE 01H*xx*). Here '*xx*' is any string.

02000
Only valid for table functions, it means that there are no more rows in the table.

38502
A special value for the case where the UDF body attempted to issue an SQL call and received an error, SQLCODE -487 (SQLSTATE 38502). because SQL is not allowed in UDFs), and chose to pass this same error back through to DB2.

Any other 38*xxx*
The function code detected an error situation. It results in a SQL error, SQLCODE -443 (SQLSTATE 38*xxx*). Here '*xxx*' is any string. Do not use 380*xx* through 384*xx* because those values are reserved by the draft extensions to the SQL92 international standard, or 385*xx* because those values are reserved by IBM.

Any other value is treated as an error situation resulting in SQLCODE -463 (SQLSTATE 39001).

- The `SQL-result-ind` is negative.

This is also true if you define the function with the NOT NULL CALL option.

function-name This argument is set by DB2 before calling the UDF. It is the qualified function name, passed from DB2 to the UDF code. This variable takes the form of a VARCHAR(27) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(27). See “How the SQL Data Types are Passed to a UDF” on page 297 for more information.

The form of the function name that is passed is:

```
<schema-name>.<function-name>
```

The parts are separated by a period. Two examples are:

```
PABLO.BLOOP       WILLIE.FINDSTRING
```

This form enables you to use the same UDF body for multiple external functions, and still differentiate between the functions when it is invoked.

Note: Although it is possible to include the period in object names, it is not recommended. For example, if a function, `rotate` is in a schema, `obj.op`, the function name that is returned is `obj.op.rotate`, and it is not obvious if the schema name is `obj` or `obj.op`.

specific-name This argument is set by DB2 before calling the UDF. It is the specific name of the function passed from DB2 to the UDF code. This variable takes the form of a VARCHAR(18) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(18). See “How the SQL Data Types are Passed to a UDF” on page 297 for more information. Two examples are:

```
willie_find_feb95       SQL9507281052440430
```

This first value is provided by the user in his CREATE FUNCTION statement. The second is a value generated by DB2 when the user does not specify a value.

As with the *function-name* argument, the reason for passing this value is to give the UDF the means of distinguishing exactly which specific function is invoking it.

diagnostic-message This argument is set by the UDF before returning to DB2. The UDF can use this argument to insert a message text in a DB2 message. It takes the form of a VARCHAR(70) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(70). See “How the SQL Data Types are Passed to a UDF” on page 297 for more information.

When the UDF returns either an error or a warning, using the *SQL-state* argument described above, it can include descriptive information here. DB2 includes this information as a token in its message.

DB2 sets the first character to null before calling the UDF. Upon return, it treats the string as a C null-terminated string. This string will be included in the SQLCA as a token for the error condition. At least the first 17 characters in this string will appear in the SQLCA or DB2 CLP message. However, the actual number of characters which will appear depends on the lengths of the other tokens. Avoid using `X'FF'` in the text since this character is used to delimit tokens in the SQLCA.

The UDF code should not change more than the VARCHAR(70) buffer which is passed to it. DB2 will attempt to determine if the UDF body has written beyond the end of this buffer by a few characters, SQLCODE -450 (SQLSTATE 39501). However, an overwrite by the UDF can cause unpredictable results or an abend, as it may not be detected by DB2.

DB2 assumes that any message tokens returned from the UDF to DB2 are in the same code page as the database. Your UDF should ensure that If this is the case. If you use the 7-bit invariant ASCII subset, your UDF can return the message tokens in any code page.

scratchpad

This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specified the SCRATCHPAD keyword. This argument is a structure, exactly like the structure used to pass a value of any of the LOB data types, with the following elements:

- An INTEGER containing the length of the scratchpad (always 100)
- The actual scratchpad (100 bytes in length), initialized to all binary 0’s before the first call to the UDF, and never looked at or modified by DB2 thereafter. If a scalar UDF which uses a scratchpad is referenced in a subquery, DB2 may decide to refresh the scratchpad between invocations of the subquery. This refresh occurs after a *final-call* is made, if FINAL CALL is specified for the UDF.

  For table functions, the scratchpad is initialized as above on each OPEN call to the UDF, and never looked at or modified by DB2 thereafter unless another OPEN call occurs. (This can happen if the UDF is used in a subquery.)

The scratchpad can be mapped in your UDF using the same type as either a CLOB or a BLOB, since the argument passed has the same structure. See “How the SQL Data Types are Passed to a UDF” on page 297 for more information.

Ensure your UDF code does not make changes outside of the scratchpad buffer. DB2 attempts to determine if the UDF body has written beyond the end of this buffer by a few characters, SQLCODE -450 (SQLSTATE 39501), but a major overwrite by the

UDF can cause unpredictable results, or an abend, and may not result in a graceful failure by DB2.

call-type

This argument is set by DB2 before calling the UDF. The argument is only present if the CREATE FUNCTION statement for the UDF specified the FINAL CALL keyword. It follows the *scratchpad* argument; or the *diagnostic-message* argument if the *scratchpad* argument is not present. This argument takes the form of an INTEGER value. Ensure that this argument definition in the UDF is appropriate for INTEGER. See "How the SQL Data Types are Passed to a UDF" on page 297 for more information.

For scalar functions it contains:

-1 This is the *first call* to the UDF for this statement. A first call is a *normal call* in that all SQL argument values are passed. The only difference is that on a first call, in cases where a scratchpad is also passed, the scratchpad is set to binary zeros.

Note that if SCRATCHPAD is specified in the CREATE FUNCTION statement, but FINAL CALL is not, the *call-type* argument is not passed to the UDF. If it is important for the UDF to identify this first call situation, it must do so by some other means. For example, it could key off the all binary zeros state of the scratchpad.

0 This is a *normal call*, that is, all the normal input argument values are passed. If a scratchpad is also passed, it is untouched from the previous call.

1 This is a *final call*, that is, no *SQL-argument* or *SQL-argument-ind* values are passed, and attempts to examine these values may cause unpredictable results. If a scratchpad is also passed, it is untouched from the previous call.

A UDF is expected to release system resources (for example, memory) acquired during the first call and succeeding normal calls. On a final call, a UDF should not return any answer using the *SQL-result* or *SQL-result-ind* arguments. Both of these are ignored by DB2 upon return from a final call to a UDF.

However, on a final call, the UDF may set the *SQL-state* and *diagnostic-message* arguments. These arguments are handled the same way as on other calls to the UDF.

If a UDF that is used in a subquery both uses a scratchpad and requires a final call, DB2 may decide to make a final call (and then refresh the scratchpad) between invocations of the subquery.

For table functions it contains:

-1 This is the *OPEN call* to the UDF for this statement. The scratchpad (if any) is set to binary zeros when the UDF is called. All argument values are passed, and the UDF should do whatever one-time initialization actions are required, but should not return a row of data to DB2.

Remember that the FINAL CALL option is mandatory for table functions, but that SCRATCHPAD is not.

0 This is a *FETCH call*, that is, all the normal input argument values are passed, same as on the OPEN CALL, and the UDF is expected to return a row or the end-of-table condition ('02000' returned as SQLSTATE to DB2). If a scratchpad is also passed, it is untouched from the previous call.

1 This is a *CLOSE call*, that is, no *SQL-argument* or *SQL-argument-ind* values are passed, and attempts to examine these values may cause unpredictable results. If a scratchpad is also passed, it is untouched from the previous call.

A UDF is expected to release system resources (for example, memory) acquired during the OPEN call and succeeding FETCH calls. On a final call, a table UDF should not return any result row to DB2. In fact, the *SQL-result* and *SQL-result-ind* arguments are ignored by DB2 upon return from a CLOSE call to a UDF.

However, on a CLOSE call, the UDF may set the *SQL-state* and *diagnostic-message* arguments. These arguments are handled the same way as on other calls to the UDF.

If a table function UDF is used in a subquery, the table function should expect 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

This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specifies the DBINFO keyword. The argument is a structure whose C language definition is contained in the header file `sqludf.h` discussed in “The UDF Include File: sqludf.h” on page 305. It contains the following elements:

1. Data base name length (dbnamelen)

   The length of *data base name* below. This field is an unsigned short integer.

2. Data base name (dbname)

The name of the currently connected database. This field is a long identifier of 128 characters. The *data base name length* field described above identifies the actual length of this field. It does not contain a null terminator or any padding.

3. Application Authorization ID Length (authidlen)

   The length of *application authorization ID* below. This field is an unsigned short integer.

4. Application authorization ID (authid)

   The application run time authorization ID. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *application authorization ID length* field described above identifies the actual length of this field.

5. Database code page (codepg)

   This is an union of two 48-byte long structures; one is used by DB2 Universal Database, the other is reserved for future use. The DB2 Universal Database structure contains the following fields:

   a. SBCS. Single byte code page, an unsigned long integer.
   b. DBCS. Double byte code page, an unsigned long integer.
   c. COMP. Composite code page, an unsigned long integer.
   d. DUMMY. Reserved for later use, a char of length 36.

6. Schema name length (tbschemalen)

   The length of *schema name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

7. Schema name (tbschema)

   Schema for the *table name* below. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *schema name length* field described above identifies the actual length of this field.

8. Table name length (tbnamelen)

   The length of the *table name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

9. Table name (tbname)

   This is the name of the table being updated or inserted. This field is set only if the UDF reference is the right-hand side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *table name length* field described above, identifies the actual length of this field. The *schema name* field above, together with this field form the fully qualified table name.

10. Column name length (colnamelen)

    Length of *column name* below. It contains a 0 (zero) if a column name is not passed. This field is an unsigned short integer.

11. Column name (colname)

    Under the exact same conditions as for table name, this field contains the name of the column being updated or inserted; otherwise not predictable. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *column name length* field described above, identifies the actual length of this field.

12. Version/Release number (ver_rel)

    An 8 character field that identifies the product and its version, release, and modification level with the format *pppvvrrm* where:

    - *ppp* identifies the product as follows:

DSN	DB2 for MVS/ESA or OS/390
ARI	SQL/DS
QSQ	DB2 for AS/400
SQL	DB2 Universal Database

    - *vv* is a two digit version identifier.
    - *rr* is a two digit release identifier.
    - *m* is a one digit modification level identifier.

13. Platform (platform)

    The following identifies the operating platform for the application server:

SQLUDF_PLATFORM_OS2	OS/2
SQLUDF_PLATFORM_AIX	AIX
SQLUDF_PLATFORM_NT	NT
SQLUDF_PLATFORM_HP	HP-UX
SQLUDF_PLATFORM_SUN	Solaris
SQLUDF_PLATFORM_SNI	Siemens Nixdorf
SQLUDF_PLATFORM_UNKNOWN	Unknown platform

For additional platforms that are not contained in the above list, see the contents of the `sqludf.h` file.

14. Number of table function column list entries (numtfcol)

    The number of non-zero entries in the table function column list specified in the *table function column list* field below.

15. Reserved field (resd1)

    This field is for future use. It is defined as 24 characters long.

16. Table function column list (tfcolumn)

    If a table function is defined, this field is a pointer to an array containing 1 000 short integers that is dynamically allocated by DB2. If a table function is not defined, this pointer is null.

    This field is used only for table functions. Only the first *n* entries, where *n* is specified in the *number of table function column list entries* field are of interest. *n* may be equal to 0, and is less than or equal to the number of result columns defined for the function in the RETURNS TABLE(...) clause of the CREATE FUNCTION statement. The values correspond to the numbers of the columns which this 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, and the values may be in any order. Note that *n* could be equal to zero, that is, the very first array element might be zero, for a statement similar to `SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ`, where no actual values are needed.

    This array represents an opportunity for optimization. The UDF need not return all values for all the result columns of the table function, only those needed in the particular context, and these are the columns identified (by number) in the array. Since this optimization may complicate the UDF logic in order to gain the performance benefit, the UDF can optimally choose to return every defined column.

17. Reserved field (resd2)

    This field is for future use. It is defined as 24 characters long.

Summary of UDF Argument Use

The following is a summary of the arguments described above, and how you use them in the interface between DB2 and an external UDF.

For scalar functions are:

- *SQL-argument*.

  This argument passes the values identified in the function reference from DB2 to the UDF. There is one of these arguments for each SQL argument.

- *SQL-result*.

  This argument passes the result value generated by the UDF back to DB2 and to the SQL statement where the function reference occurred.

- *SQL-argument-ind*.

  This argument corresponds positionally to *SQL-argument*, and tells the UDF whether or not a particular argument is null. There is one of these for each *SQL-argument*.

- *SQL-result-ind*.

  This argument is used by the UDF to report back to DB2 whether the function result in *SQL-result* contains nulls.

- *SQL-state* and *diagnostic-message*.

  These arguments are used by the UDF to signal exception information back to DB2.

- *function-name* and *specific-name*.

  These arguments are used by DB2 to pass the identity of the referenced function to the UDF.

- *scratchpad* and *call-type*.

  These arguments are used by DB2 to manage the saving of UDF state between calls. The *scratchpad* is created and initialized by DB2 and thereafter managed by the UDF. DB2 signals the type of call to the UDF using the *call-type* argument.

- *dbinfo*.

  A structure passed by DB2 to the UDF containing additional information.

A table function logically returns a table to the SQL statement that references it, but the physical interface between DB2 and the table function is row by row.

For table functions, the arguments are:

- *SQL-argument*.

  This argument passes the values identified in the function reference from DB2 to the UDF. The argument has the same value for FETCH calls as it did for the OPEN call. There is one of these for each SQL argument.

- *SQL-result*.

  This argument is used to pass back the individual column values for the row being returned by the UDF. There is one of these arguments for each result column value defined in the RETURNS TABLE (...) clause of the CREATE FUNCTION statement.

- *SQL-argument-ind*.

  This argument corresponds positionally to *SQL-argument values*, and tells the UDF whether the particular argument is null. There is one of these for each SQL argument.

- *SQL-result-ind*.

  This argument is used by the UDF to report back to DB2 whether the individual column values returned in the table function output row is null. It corresponds positionally to the *SQL-result* argument.

- *SQL-state* and *diagnostic-message*.

  These arguments are used by the UDF to signal exception information and the end-of-table condition back to DB2.

- *function-name* and *specific-name*.

  These arguments are used by DB2 to pass the identity of the referenced function to the UDF.

- *scratchpad* and *call-type*.

  These arguments are used by DB2 to manage the saving of UDF state between calls. The *scratchpad* is created and initialized by DB2 and thereafter managed by the UDF. DB2 signals the type of call to the UDF using the *call-type* argument. For table functions these call types are OPEN, FETCH, CLOSE.

- *dbinfo*.

  This is a structure passed by DB2 to the UDF containing additional information.

Observe that the normal outputs of the UDF, *SQL-result*, *SQL-result-ind*, and *SQL-state*, are returned to DB2 using arguments passed from DB2 to the UDF. Indeed, the UDF is written not to return anything in the functional sense (that is, the function's return type is void). See the `void` definition and the `return` statement in the following example:

```
#include ...
 void SQL_API_FN divid(
      ... arguments ... )
{
      ... UDF body ...
      return;
}
```

In the above example, `SQL_API_FN` is a macro that specifies the calling convention for a function that may vary across each supported operating platform. This macro is required when you write stored procedures or UDFs.

For programming examples of UDFs, see "Examples of UDF Code" on page 322.

How the SQL Data Types are Passed to a UDF

This section identifies the valid types, for both UDF parameters and result, and specifies for each how the corresponding argument should be defined in your C or C++ language UDF. Note that if you use the `sqludf.h` include file and the types defined there, you can automatically generate language variables and structures that are correct for the different data types. This include file is discussed in "The UDF Include File: sqludf.h" on page 305.

It is the data type for each function parameter defined in the CREATE FUNCTION statement that governs the format for argument values. Promotions from the argument data type may be needed to get the value in that format. Such promotions are performed automatically by DB2 on the argument values; argument promotion is discussed in the *SQL Reference*.

For the function result, it is the data type specified in the CAST FROM clause of the CREATE FUNCTION statement that defines the format. If no CAST FROM clause is present, then the data type specified in the RETURNS clause defines the format.

In the following example, the presence of the CAST FROM clause means that the UDF body returns a SMALLINT and that DB2 casts the value to INTEGER before passing it along to the statement where the function reference occurs:

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

In this case the UDF must be written to generate a SMALLINT, as defined below. Note that the CAST FROM data type must be *castable* to the RETURNS data type, so one cannot just arbitrarily choose another data type. Casting between data types is discussed in the *SQL Reference*.

The following is a list of the SQL types and their C language representations. It includes information on whether each type is valid as a parameter or a result. Also included are examples of how the types could appear as an argument definition in your C or C++ language UDF:

- SMALLINT

  Valid. Represent in C as `short`.

  When defining integer UDF parameters, consider using INTEGER rather than SMALLINT as DB2 does not promote SMALLINT arguments to INTEGER. For example, suppose you define a UDF as follows:

  ```
  CREATE FUNCTION SIMPLE(SMALLINT)...
  ```

  If you invoke the SIMPLE function using INTEGER data, (`...SIMPLE(1)...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function may not perceive the reason for the message. In the above example, `1` is an INTEGER, so you can either cast it to SMALLINT or define the parameter as INTEGER.

  Example:

  ```
  short   *arg1;          /* example for SMALLINT */
  short   *arg1_null_ind; /* example for any null indicator */
  ```

- INTEGER or INT

  Valid. Represent in C as `long`.

  Example:

  ```
  long    *arg2;          /* example for INTEGER */
  ```

- DECIMAL(p,s) or NUMERIC(p,s)

  Not valid, because there is no C language representation. If you want to pass a decimal value, you must define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE), and explicitly cast the argument to this type. In the case of DOUBLE, you do not need to explicitly cast a decimal argument to a DOUBLE parameter as DB2 promotes it automatically .

  Suppose you have two columns, WAGE as DECIMAL(5,2) and HOURS as DECIMAL(4,1), and you wish to write a UDF to calculate weekly pay based on wage, number of hours worked and some other factors. The UDF could be as follows:

  ```
  CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
         RETURNS DECIMAL(7,2) CAST FROM DOUBLE
         ...;
  ```

  For the above UDF, the first two parameters correspond to the wage and number of hours. You invoke the UDF WEEKLY_PAY in your SQL select statement as follows:

  ```
  SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
  ```

  Note that no explicit casting is required because the DECIMAL arguments are castable to DOUBLE.

  Alternatively, you could define WEEKLY_PAY with CHAR arguments as follows:

  ```
  CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
         RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
         ...;
  ```

  You would invoke it as follows:

  ```
  SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
  ```

  Observe the explicit casting that is required because DECIMAL arguments are not promotable to VARCHAR.

  An advantage of using floating point parameters is that it is easier to perform arithmetic on the values in the UDF; an advantage of using character parameters is that it is always possible to exactly represent the decimal value. This is not always possible with floating point.

- REAL

  Valid. Represent in C as `float`.

  Example:

  ```
  float *result;          /* example for REAL */
  ```

- DOUBLE or DOUBLE PRECISION or FLOAT

  Valid. Represent in C as `double`.

  Example:

  ```
  double  *result;        /* example for DOUBLE   */
  ```

- CHAR(n) or CHARACTER(n) with or without the FOR BIT DATA modifier.

  Valid. Represent in C as `char...[n+1]` (this is a C null-terminated string, the last character is a null, that is X'00').

  For a CHAR(n) parameter, DB2 always moves n bytes of data to the buffer and sets the *n+1* byte to null. For a RETURNS CHAR(n) value, DB2 always takes the n bytes and ignores the *n+1* byte. For this RETURNS CHAR(n) case, you are warned against the inadvertent inclusion of a null-character in the first n characters. DB2 will not recognize this as anything but a normal part of the data, and it might later on cause seemingly anomalous results if it was not intended.

  If FOR BIT DATA is specified, exercise caution about using the normal C string handling functions in the UDF. Many of these functions look for a null to delimit the string, and the null-character (X'00') could be a legitimate character in the middle of the data value.

  When defining character UDF parameters, consider using VARCHAR rather than CHAR as DB2 does not promote VARCHAR arguments to CHAR. For example, suppose you define a UDF as follows:

  ```
  CREATE FUNCTION SIMPLE(INT,CHAR(1))...
  ```

  If you invoke the SIMPLE function using VARCHAR data, (`...SIMPLE(1,'A')...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function may not perceive the reason for the message. In the above example, `'A'` is VARCHAR, so you can either cast it to CHAR or define the parameter as VARCHAR.

  Example:

  ```
  char     arg1[14];      /* example for CHAR(13)   */
  char    *arg1;          /* also perfectly acceptable */
  ```

- VARCHAR(n) FOR BIT DATA or LONG VARCHAR with or without the FOR BIT DATA modifier.

  Valid. Represent in C as a structure similar to:

  ```
  struct sqludf_vc_fbd
  {
     unsigned short length;         /* length of data */
     char           data[1];        /* first char of data */
  };
  ```

  The [1] is merely to indicate an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

  These values are not represented as C null-terminated strings because the null-character could legitimately be part of the data value. The length is explicitly passed to the UDF for parameters using the structure variable `length`. For the RETURNS clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable `length`, is the actual length of the data value.

Example:

```
struct sqludf_vc_fbd *arg1;  /* example for VARCHAR(n) FOR BIT DATA */
struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
```

- VARCHAR(n) without FOR BIT DATA.

  Valid. Represent in C as `char...[n+1]`. (This is a C null-terminated string.)

  For a VARCHAR(n) parameter, DB2 will put a null in the (k+1) position, where k is the length of the particular occurrence. The C string-handling functions are thus well suited for manipulation of these values. For a RETURNS VARCHAR(n) value, the UDF body must delimit the actual value with a null, because DB2 will determine the result length from this null character.

  Example:

  ```
  char     arg2[51];      /* example for VARCHAR(50)   */
  char    *result;        /* also perfectly acceptable */
  ```

- GRAPHIC(n)

  Valid. Represent in C as `sqldbchar[n+1]`. (This is a null-terminated graphic string). Note that you can use `wchar_t[n+1]` on platforms where `wchar_t` is defined to be 2 bytes in length; however, `sqldbchar` is recommended. Refer to “Selecting the wchar_t or sqldbchar Data Type” on page 441 for more information on these two data types.

  For a GRAPHIC(n) parameter, DB2 moves n double-byte characters to the buffer and sets the following two bytes to null. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option described in “The WCHARTYPE Precompiler Option” on page 441. For a RETURNS GRAPHIC(n) value, DB2 always takes the n double-byte characters and ignores the following bytes.

  When defining graphic UDF parameters, consider using VARGRAPHIC rather than GRAPHIC as DB2 does not promote VARGRAPHIC arguments to GRAPHIC. For example, suppose you define a UDF as follows:

  ```
  CREATE FUNCTION SIMPLE(GRAPHIC)...
  ```

  If you invoke the SIMPLE function using VARGRAPHIC data, (`...SIMPLE('` *`graphic_literal`*`')...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function may not understand the reason for this message. In the above example, *`graphic_literal`* is a literal DBCS string that is interpreted as VARGRAPHIC data, so you can either cast it to GRAPHIC or define the parameter as VARGRAPHIC.

  Example:

  ```
  sqldbchar   arg1[14];      /* example for GRAPHIC(13)   */
  sqldbchar  *arg1;          /* also perfectly acceptable */
  ```

- VARGRAPHIC(n)

  Valid. Represent in C as sqldbchar[n+1]. (This is a null-terminated graphic string). Note that you can use wchar_t[n+1] on platforms where wchar_t is defined to be 2 bytes in length; however, sqldbchar is recommended. Refer to “Selecting the wchar_t or sqldbchar Data Type” on page 441 for more information on these two data types.

  For a VARGRAPHIC(n) parameter, DB2 will put a graphic null in the (k+1) position, where k is the length of the particular occurrence. A graphic null refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option described in “The WCHARTYPE Precompiler Option” on page 441. For a RETURNS VARGRAPHIC(n) value, the UDF body must delimit the actual value with a graphic null, because DB2 will determine the result length from this graphic null character.

  Example:

  ```
  sqldbchar   args[51],    /* example for VARGRAPHIC(50) */
  sqldbchar   *result,     /* also perfectly acceptable  */
  ```

- LONG VARGRAPHIC

  Valid. Represent in C as a structure:

  ```
  struct sqludf_vg
  {
     unsigned short length;         /* length of data */
     sqldbchar      data[1];        /* first char of data */
  };
  ```

  Note that in the above structure, you can use wchar_t in place of sqldbchar on platforms where wchar_t is defined to be 2 bytes in length, however, the use of sqldbchar is recommended. Refer to “Selecting the wchar_t or sqldbchar Data Type” on page 441 for more information on these two data types.

  The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed. Because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

  These are not represented as null-terminated graphic strings. The length, in double-byte characters, is explicitly passed to the UDF for parameters using the structure variable length. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option described in “The WCHARTYPE Precompiler Option” on page 441. For the RETURNS clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable length, is the actual length of the data value, in double byte characters.

  Example:

  ```
  struct sqludf_vg *arg1;  /* example for VARGRAPHIC(n)   */
  struct sqludf_vg *result; /* also for LONG VARGRAPHIC   */
  ```

- DATE

  Valid. Represent in C same as CHAR(10), that is as `char...[11]`. The date value is always passed to the UDF in ISO format: `yyyy-mm-dd`.

  Example:

  ```
  char     arg1[11];      /* example for DATE       */
  char    *result;        /* also perfectly acceptable */
  ```

- TIME

  Valid. Represent in C same as CHAR(8), that is, as `char...[9]`. The time value is always passed to the UDF in ISO format: `hh.mm.ss`.

  Example:

  ```
  char    *arg;           /* example for DATE          */
  char     result[9];     /* also perfectly acceptable */
  ```

- TIMESTAMP

  Valid. Represent in C same as CHAR(26), that is. as `char...[27]`. The timestamp value is always passed with format: `yyyy-mm-dd-hh.mm.ss.nnnnnn`.

  Example:

  ```
  char     arg1[27];      /* example for TIMESTAMP  */
  char    *result;        /* also perfectly acceptable */
  ```

- BLOB(n) and CLOB(n)

  Valid. Represent in C as a structure:

  ```
  struct sqludf_lob
  {
      unsigned long length;      /* length in bytes */
      char          data[1];      /* first byte of lob */
  };
  ```

  The [1] merely indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

  These are not represented as C null-terminated strings. The length is explicitly passed to the UDF for parameters using the structure variable `length`. For the RETURNS clause, the length that is passed back to the UDF, is the length of the buffer. What the UDF body must pass back, using the structure variable `length`, is the actual length of the data value.

  Example:

  ```
  struct sqludf_lob *arg1;  /* example for BLOB(n), CLOB(n) */
  struct sqludf_lob *result;
  ```

- DBCLOB(n)

  Valid. Represent in C as a structure:

```
struct sqludf_lob
{
    unsigned long length;      /* length in graphic characters */
    sqldbchar data[1];         /* first byte of lob */
};
```

Note that in the above structure, you can use `wchar_t` in place of `sqldbchar` on platforms where `wchar_t` is defined to be 2 bytes in length, however, the use of `sqldbchar` is recommended. Refer to “Selecting the wchar_t or sqldbchar Data Type” on page 441 for more information on these two data types.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

These are not represented as null-terminated graphic strings. The length is explicitly passed to the UDF for parameters using the structure variable `length`. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option described in “The WCHARTYPE Precompiler Option” on page 441. For the RETURNS clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable `length`, is the actual length of the data value, with all of these lengths expressed in double byte characters.

Example:

```
struct sqludf_lob *arg1;  /* example for DBCLOB(n) */
struct sqludf_lob *result;
```

- Distinct Types

  Valid or invalid depending on the base type. Distinct types will be passed to the UDF in the format of the base type of the UDT, so may be specified if and only if the base type is valid.

  Example:

```
struct sqludf_lob *arg1;  /* for distinct type based on BLOB(n) */
double            *arg2;  /* for distinct type based on DOUBLE  */
char              res[5]; /* for distinct type based on CHAR(4) */
```

- Distinct Types AS LOCATOR , or any LOB type AS LOCATOR

  The AS LOCATOR type modifier is valid only in UDF parameter and result definitions. It may only be used to modify the LOB types or any distinct type that is based on a LOB type. If you specify the type modifier, a four byte locator is passed to the UDF rather the entire LOB value.

  Example:

```
sqludf_locator      *arg1;  /* locator argument */
sqludf_locator      *result; /* locator result */
```

The type `udf_locator` is defined in the header file sqludf.h, which is discussed in “The UDF Include File: sqludf.h” on page 305. The use of these locators is discussed in “Using LOB Locators as UDF Parameters or Results” on page 315.

The UDF Include File: sqludf.h

This include file contains structures, definitions and values which are useful when writing your UDF. Its use is optional, however, and in the sample UDFs shown in "Examples of UDF Code" on page 322, some examples use the include file. When compiling your UDF, you need to reference the directory which contains this file. This directory is `sqllib/include`.

The `sqludf.h` include file is self-describing. Following is a brief summary of its content:

1. Structure definitions for the passed arguments which are structures:
   - VARCHAR FOR BIT DATA arguments and result
   - LONG VARCHAR (with or without FOR BIT DATA) arguments and result
   - LONG VARGRAPHIC arguments and result
   - All the LOB types, SQL arguments and result
   - The scratchpad
   - The dbinfo structure.
2. C language type definitions for all the SQL data types, for use in the definition of UDF arguments corresponding to SQL arguments and result having the data types. These are the definitions with names SQLUDF_x and SQLUDF_x_FBD where x is a SQL data type name, and FBD represents For Bit Data.

   Also included is a C language type for an argument or result which is defined with the AS LOCATOR appendage.
3. Definition of C language types for the *scratchpad* and *call-type* arguments, with an `enum` type definition of the *call-type* argument.
4. Macros for defining the standard *trailing* arguments, both with and without the inclusion of *scratchpad* and *call-type* arguments. This corresponds to the presence and absence of SCRATCHPAD and FINAL CALL keywords in the function definition. These are the *SQL-state*, *function-name*, *specific-name*, *diagnostic-message*, *scratchpad* and *call-type* UDF invocation arguments defined in "The Arguments Passed from DB2 to a UDF" on page 286. Also included are definitions for referencing these constructs, and the various valid SQLSTATE values.
5. Macros for testing whether the SQL arguments are null.
6. Function prototypes for the APIs which can be used to manipulate LOB values by means of LOB locators passed to the UDF.

Some of the UDF examples in the next section illustrate the inclusion and use of `sqludf.h`.

Writing OLE Automation UDFs

OLE (Object Linking and Embedding) automation is part of the OLE 2.0 architecture from Microsoft Corporation. With OLE automation, your applications, regardless of the language in which they are written, can expose their properties and methods in OLE automation objects. These properties and methods can be accessed by other applications through OLE automation. The applications exposing the properties and methods are called OLE automation servers or objects, and the applications that access those properties and methods are called OLE automation controllers. OLE automation servers are COM components (objects) that implement the OLE IDispatch interface. An OLE automation controller is a COM client that communicates with the automation server through its IDispatch interface. COM (Component Object Model) is the foundation of OLE. For OLE automation UDFs, DB2 acts as an OLE automation controller. Through this mechanism, DB2 can invoke methods of OLE automation objects as external UDFs.

Note that this section assumes that you are familiar with OLE automation terms and concepts. This book does not present any introductory OLE material. For an overview of OLE automation, refer to *Microsoft Corporation: The Component Object Model Specification*, October 1995. For details on OLE automation, refer to *OLE Automation Programmer's Reference*, Microsoft Press, 1996, ISBN 1-55615-851-3.

Creating and Registering OLE Automation UDFs

OLE automation UDFs are implemented as public methods of OLE automation objects. The OLE automation objects must be externally creatable by an OLE automation controller, in this case DB2, and support late binding (also called IDispatch-based binding). OLE automation objects must be registered in the Windows registration database (registry) with a class identifier (CLSID), and optionally, an OLE programmatic ID (progID) to identify the automation object.
The progID can identify an in-process (.DLL) or local (.EXE) OLE automation server, or a remote server through DCOM (Distributed COM). OLE automation UDFs can be scalar functions or table functions.

After you code an OLE automation object, you need to register the methods of the object as UDFs using the SQL CREATE FUNCTION statement. Registering an OLE automation UDF is very similar to registering any external C or C++ UDF. OLE automation UDFs are registered by using the LANGUAGE OLE clause. The external name consists of the OLE progID identifying the OLE automation object and the method name separated by ! (exclamation mark):

```
CREATE FUNCTION bcounter () RETURNS INTEGER
  EXTERNAL NAME 'bert.bcounter!increment'
  LANGUAGE OLE
  FENCED
  SCRATCHPAD
  FINAL CALL
  NOT DETERMINISTIC
  NULL CALL
  PARAMETER STYLE DB2SQL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

The calling conventions for OLE method implementations are identical to the conventions for functions written in C or C++. An implementation of the above method in the BASIC language looks like the following (notice, that in BASIC the parameters are by default defined as call by reference):

```
Public Sub increment(output As Long, _
                     indicator As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                     sqlmsg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

Object Instance and Scratchpad Considerations

OLE automation UDFs (methods of OLE automation objects) are applied on instances of OLE automation objects. DB2 creates an object instance for each UDF reference in an SQL statement. An object instance can be reused for subsequent method invocations of the UDF reference in an SQL statement, or the instance can be released after the method invocation and a new instance is created for each subsequent method invocation. The proper behavior can be specified with the SCRATCHPAD option in the SQL CREATE FUNCTION statement. For the LANGUAGE OLE clause, the SCRATCHPAD option has the additional semantic compared to C or C++, that a single object instance is created and reused for the entire query, whereas if NO SCRATCHPAD is specified, a new object instance may be created each time a method is invoked. Separate instances are created for each UDF reference in an SQL statement.

Using the scratchpad allows a method to maintain state information in instance variables of the object, across function invocations. It also increases performance as an object instance is only created once and then reused for subsequent invocations.

How the SQL Data Types are Passed to an OLE Automation UDF

DB2 handles the type conversions between SQL types and OLE automation types. The following table summarizes the supported data types and how they are mapped. The mapping of OLE automation types to data types of the implementing programming language, such as BASIC or C/C++, is described in Table 12 on page 309.

Table 11. Mapping of SQL and OLE Automation Datatypes

SQL Type	**OLE Automation Type**	**OLE Automation Type Description**
SMALLINT	short	16-bit signed integer
INTEGER	long	32-bit signed integer
REAL	float	32-bit IEEE floating-point number
FLOAT or DOUBLE	double	64-bit IEEE floating-point number
DATE	DATE	64-bit floating-point fractional number of days since December 30, 1899
TIME	DATE	
TIMESTAMP	DATE	
CHAR(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARCHAR(*n*)	BSTR	
LONG VARCHAR	BSTR	
CLOB(*n*)	BSTR	
GRAPHIC(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARGRAPHIC(*n*)	BSTR	
LONG GRAPHIC	BSTR	
DBCLOB(*n*)	BSTR	
CHAR(*n*)[1]	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)
VARCHAR(*n*)[1]	SAFEARRAY[unsigned char]	
LONG VARCHAR[1]	SAFEARRAY[unsigned char]	
BLOB(*n*)	SAFEARRAY[unsigned char]	

Note:

1. With FOR BIT DATA specified

Data passed between DB2 and OLE automation UDFs is passed as call by reference. SQL types such as DECIMAL or LOCATORS, or OLE automation types such as boolean or CURRENCY that are not listed in the table are not supported. Character and graphic data mapped to BSTR is converted from the database code page to the UCS-2 (also known as Unicode, IBM code page 13488) scheme. Upon return, the data is converted back to the database code page. These conversions occur regardless of the database code page. If code page conversion tables to convert from the database code page to UCS-2 and from UCS-2 to the database code page are not installed, you receive an SQLCODE -332 (SQLSTATE 57017).

Implementing OLE Automation UDFs in BASIC and C++

You can implement OLE automation UDFs in any language. In this section, you are shown how to implement OLE automation UDFs using BASIC or C++ as two sample languages.

Table 12 shows the mapping of the various SQL data types to the intermediate OLE automation data types, and the data types in the language of interest (BASIC or C++). OLE data types are language independent, (that is, Table 11 on page 308 holds true for all languages).

Table 12. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types

		UDF Language	
SQL Type	**OLE Automation Type**	**BASIC Type**	**C++ Type**
SMALLINT	short	Integer	short
INTEGER	long	Long	long
REAL	float	Single	float
FLOAT or DOUBLE	double	Double	double
DATE, TIME, TIMESTAMP	DATE	Date	DATE
CHAR(*n*), VARCHAR(*n*), LONG VARCHAR, CLOB(*n*)	BSTR	String	BSTR
GRAPHIC(*n*), VARGRAPHIC(*n*), LONG GRAPHIC, DBCLOB(*n*)	BSTR	String	BSTR
CHAR(*n*)[1] , VARCHAR(*n*)[1] , LONG VARCHAR[1] , BLOB(*n*)	SAFEARRAY[unsigned char]	byte()	SAFEARRAY

Note:

1. With FOR BIT DATA specified

OLE Automation UDFs in BASIC

To implement OLE automation UDFs in BASIC you need to use the BASIC data types corresponding to the SQL data types mapped to OLE automation types.

The BASIC declaration of the previously registered increment OLE automation UDF looks like the following:

```
Public Sub increment(output As Long, _
                     indicator As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                     sqlmsg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

An OLE automation UDF can be a scalar function or a table function. The following example registers an OLE automation table function that returns header information of e-mail messages in a mailbox together with the first 30 characters of the message text.

```
CREATE FUNCTION mail ()
  RETURNS TABLE (TIMERECEIVED DATE,
                 SUBJECT VARCHAR(15),
                 SIZE INTEGER,
                 TEXT VARCHAR (30))
  EXTERNAL NAME 'tfmail.header!list'
  LANGUAGE OLE
  FENCED
  SCRATCHPAD
  FINAL CALL
  NOT DETERMINISTIC
  NULL CALL
  DISALLOW PARALLEL
  PARAMETER STYLE DB2SQL
  NO SQL
  NO EXTERNAL ACTION;
```

The implementation of the table function is described in “Example: Mail OLE Automation Table Function in BASIC” on page 345.

OLE Automation UDFs in C++

Table 12 on page 309 shows the C++ data types that correspond to the SQL data types and how they map to OLE automation types.

The C++ declaration of the above registered OLE automation UDF is as follows:

```
STDMETHODIMP Ccounter::increment (long   *output,
                                  short  *indicator,
                                  BSTR   *sqlstate,
                                  BSTR   *fname,
                                  BSTR   *fspecname,
                                  BSTR   *sqlmsg,
                                  SAFEARRAY **scratchpad,
                                  long   *calltype );
```

OLE supports type libraries that describe the properties and methods of OLE automation objects. Exposed objects, properties, and methods are described in the Object Description Language (ODL). The ODL description of the above C++ method is as follows:

```
HRESULT increment ([out]    long  *output,
                   [out]    short *indicator,
                   [out]    BSTR  *sqlstate,
                   [in]     BSTR  *fname,
                   [in]     BSTR  *fspecname,
                   [out]    BSTR  *sqlmsg,
                   [in,out] SAFEARRAY (unsigned char) *scratchpad,
                   [in]     long *calltype);
```

The ODL description allows the specification whether a parameter is an input (in), output (out) or input/output (in,out) parameter. For an OLE automation UDF, the UDF input parameters and its input indicators are specified as [in] parameters, and UDF output parameters and its output indicators as [out] parameters. For the UDF trailing arguments, sqlstate is an [out] parameter, function name and function specific name are [in] parameters, scratchpad is an [in,out] parameter, and call type is an [in] parameter.

Scalar functions contain one output parameter and output indicator, whereas table functions contain multiple output parameters and output indicators corresponding to the RETURN columns of the CREATE FUNCTION statement.

OLE automation defines the BSTR data type to handle strings. BSTR is defined as a pointer to OLECHAR: typedef OLECHAR *BSTR. For allocating and freeing BSTRs, OLE imposes the rule, that the callee frees a BSTR passed in as a by-reference parameter before assigning the parameter a new value. This rule means the following for DB2 and OLE automation UDFs. The same rule applies for one-dimensional byte arrays which are received by the callee as SAFEARRAY**:

- [in] parameters: DB2 allocates and frees [in] parameters.
- [out] parameters: DB2 passes in a pointer to NULL. The [out] parameter must be allocated by the callee and is freed by DB2.
- [in,out] parameters: DB2 initially allocates [in,out] parameters. They can be freed and re-allocated by the callee. As is true for [out] parameters, DB2 frees the final returned parameter.

All other parameters are passed as pointers. DB2 allocates and manages the referenced memory.

OLE automation provides a set of data manipulation functions for dealing with BSTRs and SAFEARRAYs. The data manipulation functions are described in the *OLE Automation Programmer's Reference*.

The following C++ UDF returns the first 5 characters of a CLOB input parameter:

```
// UDF DDL: CREATE FUNCTION crunch (clob(5k)) RETURNS char(5)

STDMETHODIMP Cobj::crunch (BSTR *in,          // CLOB(5K)
                           BSTR *out,         // CHAR(5)
                           short indicator1,  // input indicator
                           short indicator2,  // output indicator
                           BSTR *sqlstate,    // pointer to NULL
                           BSTR *fname,       // pointer to function name
                           BSTR *fspecname,   // pointer to specific name
                           BSTR *msgtext)     // pointer to NULL
  {
     // Allocate BSTR of 5 characters
     // and copy 5 characters of input parameter

     // out is an [out] parameter of type BSTR, that is,
     // it is a pointer to NULL and the memory does not have to be freed.
     // DB2 will free the allocated BSTR.

     *out = SysAllocStringLen (*in, 5);
     return NOERROR;
  };
```

An OLE automation server can be implemented as *creatable single-use* or *creatable multi-use*. With creatable single-use, each client (that is, a DB2 fenced process) connecting with `CoGetClassObject` to an OLE automation object will use its own instance of a class factory, and run a new copy of the OLE automation server if necessary. With creatable multi-use, many clients connect to the same class factory. That is, each instantiation of a class factory is supplied by an already running copy of the OLE server, if any. If there are no copies of the OLE server running, a copy is automatically started to supply the class object. The choice between single-use and multi-use OLE automation servers is yours, when you implement your automation server. A single-use server is recommended for better performance.

Scratchpad Considerations

The factors influencing whether your UDF should use a scratchpad or not are important enough to warrant this special section. Other coding considerations are discussed in "Other Coding Considerations" on page 318.

It is important that you code UDFs to be re-entrant. This is primarily due to the fact that many references to the UDF may use the same copy of the function body. In fact, these *many references* may even be in different statements or applications. However, note that functions may need or want to save state from one invocation to the next. Two categories of these functions are:

1. Functions that, to be correct, depend on saving state.

   An example of such a function is a simple *counter* function which returns a `'1'` the first time it is called, and increments the result by one each successive call. Such a function could be used to number the rows of a SELECT result:

   ```
   SELECT counter(), a, b+c, ...
     FROM tablex
     WHERE ...
   ```

   This type of function is NOT DETERMINISTIC (or VARIANT). Its output does not depend solely on the values of its SQL arguments. This *counter* function is shown in “Example: Counter” on page 330.

2. Functions where the performance can be improved by the ability to perform some initialization actions one time only.

   An example of such a function, which may be a part of a document application, is a *match* function, which returns `'Y'` if a given document contains a given string, and `'N'` otherwise:

   ```
   SELECT docid, doctitle, docauthor
     FROM docs
     WHERE match('myocardial infarction', docid) = 'Y'
   ```

   This statement returns all the documents containing the particular text string value represented by the first argument. What *match* would like to do is:

   - First time only.

     Retrieve a list of all the document IDs which contain the string `myocardial infarction` from the document application which is maintained outside of DB2. This retrieval is a costly process, so the function would like to do it only one time, and save the list somewhere handy for subsequent calls.

   - On each call.

     Use the list of document IDs saved during this first call to see if the document ID which is passed as the second argument is contained in the list.

   This particular *match* function is DETERMINISTIC (or NOT VARIANT). Its answer only depends on its input argument values. What is shown here is a function whose performance, not correctness, depends on the ability to save information from one call to the next.

Both of these needs are met by the ability to specify a SCRATCHPAD in the CREATE FUNCTION statement:

```
CREATE FUNCTION counter()
  RETURNS int ... SCRATCHPAD;
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD;
```

This SCRATCHPAD keyword tells DB2 to allocate and maintain a scratchpad for the function. DB2 initializes the scratchpad to binary zeros, and does not examine or change the content thereafter. The scratchpad is passed to the function on each invocation. The function can be reentrant, and DB2 preserves its state information in the scratchpad.

So for *counter*, the last value returned could be kept in the scratchpad. And *match* could keep the list of documents in the scratchpad if the scratchpad is big enough, or otherwise could allocate memory for the list and keep the address of the acquired memory in the scratchpad.

Because it is recognized that a UDF may want to acquire system resources, the UDF can be defined with the FINAL CALL keyword. This keyword tells DB2 to call the UDF at end-of-statement processing so that the UDF can release its system resources. In particular, since the scratchpad is of fixed size, the UDF may want to allocate memory for itself and thus uses the final call to free the memory. For example the *match* function above cannot predict how many documents will match the given text string. So a better definition for *match* is:

```
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD  FINAL CALL;
```

Note that for UDFs that use a scratchpad and are referenced in a subquery, DB2 may decide to make a final call (if the UDF is so specified) and refresh the scratchpad between invocations of the subquery. You can protect yourself against this possibility, if your UDFs are ever used in subqueries, by defining the UDF with FINAL CALL and using the call-type argument, or by always checking for the *binary zero* condition.

Table Function Considerations

A table function is an external UDF which delivers a table to the SQL in which it is referenced. A table function reference is only valid in a FROM clause of a SELECT. When using table functions, observe the following:

- Even though a table function delivers a table, the physical interface between DB2 and the UDF is one-row-at-a-time. There are three types of call made to a table function: OPEN, FETCH and CLOSE. The same *call-type* mechanism that can be used for scalar functions is used to distinguish these calls.
- The standard interface used between DB2 and user-defined scalar functions is extended to accommodate table functions. The *SQL-result* repeats for table functions, each instance corresponding to a column to be returned as defined in the RETURNS TABLE clause of the CREATE FUNCTION statement. The *SQL-result-ind* argument likewise repeats, each instance related to the corresponding *SQL-result* instance.
- Not every result column defined in the RETURNS clause of the CREATE FUNCTION statement for the table function has to be returned. The DBINFO keyword of CREATE FUNCTION, and corresponding *dbinfo* argument enable the optimization that only those columns needed for a particular table function reference need be returned.
- The individual column values returned conform in format to the values returned by scalar functions.

- The CREATE FUNCTION statement for a table function has a *CARDINALITY n* specification. This specification enables the definer to inform the DB2 optimizer of the approximate size of the result so that the optimizer can make better decisions when the function is referenced.

  Regardless of what has been specified as the CARDINALITY of a table function, exercise caution against writing a function with infinite cardinality, that is, a function that always returns a row on a FETCH call. There are many situations where DB2 expects the *end-of-table* condition, as a catalyst within its query processing. Using GROUP BY or ORDER BY are examples where this is the case. DB2 cannot form the groups for aggregation until *end-of-table* is reached, and it cannot sort until it has all the data.

Using LOB Locators as UDF Parameters or Results

You can append `AS LOCATOR` to any of the LOB data types, or any distinct types based on LOB types in a CREATE FUNCTION statement. This applies both to the parameters that are passed and the results that are returned. When this happens, DB2 does the following:

- For parameters, DB2 passes a four byte locator instead of the entire LOB value. This locator can be used in a number of ways, involving a set of special APIs (described below), to retrieve and manipulate the actual bytes. The savings are clear for the case where the UDF only needs a few bytes of the value:

Storage	Memory for the entire LOB need not be allocated.
Performance	Materializing the entire value can take a significant amount of I/O time and byte moving instructions.

- For results, returns a four-byte locator instead of the entire LOB value. Again there can be storage and performance benefits.

Do not modify the locator values this makes them unusable, and the APIs will return errors.

These special APIs can only be used in UDFs which are defined as NOT FENCED. This implies that these UDFs in test phase should not be used on a production database, because of the possibility that a UDF with bugs could cause the system harm. When operating on a test database, no lasting harm can result from the UDF if it should have bugs. When the UDF is known to be free of errors it can then be applied to the production database, and run NOT FENCED without causing difficulty.

The APIs, which follow, are defined using the function prototypes contained in the sqludf.h UDF include file:

```
extern int sqludf_length(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    long*               Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_substr(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    long                start,          /* in:  Substring start value (starts at 1) */
    long                length,         /* in:  Get this many bytes */
    unsigned char*      buffer_p,       /* in:  Read into this buffer */
    long*               Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_append(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    unsigned char*      buffer_p,       /* in:  User's data buffer */
    long                length,         /* in:  Length of data to be appended */
    long*               Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_create_locator(
    int                 loc_type,       /* in:  BLOB, CLOB or DBCLOB? */
    sqludf_locator**    Loc_p           /* out: Return a ptr to a new locator */
);
extern int sqludf_free_locator(
    sqludf_locator*     loc_p           /* in: User-supplied LOB locator value */
);
```

Figure 25. DB2 Lob Locator APIs Defined in sqludf.h

The following is a discussion of how these APIs operate. Note that all lengths are in bytes, regardless of the data type, and not in single or double-byte characters.

1. `sqludf_length()`.

   Given a LOB locator, it returns the length of the LOB value represented by the locator. The locator in question is generally a locator passed to the UDF by DB2, but could be a locator representing a result value being built (using `sqludf_append()`) by the UDF.

   Typically, a UDF uses this API when it wants to find out the length of a LOB value when it receives a locator.

   The return code passed back to the UDF by DB2 is:

   0 Success.
   -1 Locator passed to the API was freed by `sqludf_free_locator()` prior to making the call.
   -2 Call was attempted in FENCED mode UDF.
   other Invalid locator.

2. `sqludf_substr()`

   Given a LOB locator, a beginning position within the LOB, a desired length, and a pointer to a buffer, this API places the bytes into the buffer and returns the number of bytes it was able to move. (Obviously the UDF must provide a buffer large enough for the desired length.) The number of bytes moved could be shorter than the desired length, for example if you request 50 bytes beginning at position 101 and the LOB value is only 120 bytes long, the API will move only 20 bytes.

Typically, this is the API that a UDF uses when it wants to see the bytes of the LOB value, when it receives a locator.

The return code passed back to the UDF by DB2 is:

0 Success.
-1 Locator passed to the API was freed by `sqludf_free_locator()` prior to making the call.
-2 Call was attempted in FENCED mode UDF.
other Invalid locator, or other error (for example, I/O error).

3. `sqludf_append()`

   Given a LOB locator, a pointer to a data buffer which has data in it, and a length of data to append, this API appends the data to the end of the LOB value, and returns the length of the bytes appended. (Note that the length appended is always equal to the length given to append. If the entire length cannot be appended, the call to `sqludf_append()` fails with the return code of `other`.)

   Typically, this is the API that a UDF uses when the result is defined with AS LOCATOR, and the UDF is building the result value one at a time after creating the locator using `sqludf_create_locator()`. After finishing the build process in this case, the UDF moves the locator to where the result argument points.

   Note that you can also append to your input locators using this API, which might be useful from the standpoint of maximum flexibility to manipulate your values within the UDF, but this will not have any affect on any LOB values in the SQL statement, or stored in the database.

   The return code passed back to the UDF by DB2 is:

   0 Success.
   -1 Locator passed to the API was freed by `sqludf_free_locator()` prior to making the call.
   -2 Call was attempted in FENCED mode UDF.
   other Invalid locator, or other error (for example, I/O error or memory error).

4. `sqludf_create_locator()`

   Given a data type, for example `SQL_TYP_CLOB`, it creates a locator. (The data type values are defined in the external application header file `sql.h`.)

   Typically, a UDF uses this API when the UDF result is defined with AS LOCATOR, and the UDF wants to build the result value using `sqludf_append()`. Another use is to internally manipulate LOB values.

   The return code passed back to the UDF by DB2 is:

   0 Success.
   -2 Call was attempted in FENCED mode UDF.
   other Unable to create a locator.

5. `sqludf_free_locator()`

Frees the passed locator.

Typically, this is used to free locators passed to the UDF that are used in the internal manipulation of LOB values. A UDF should not need to free a locator that it returns to DB2 as a result as DB2 takes care of freeing it.

0	Success.
-2	Call was attempted for a UDF in FENCED mode.
other	Unable to create a locator.

Scenarios for Using LOB Locators

This is a brief summary of possible scenarios that show the usefulness of LOB locators. These four scenarios outline the use of locators, and show how you can reduce space requirements and increase efficiency.

1. Varying access to parts of an input LOB.

   A UDF looks at the first part of a LOB value using `sqludf_substr()`, and based on a size variable it finds there, it may want to read just a few bytes from anywhere in the 100 million byte LOB value, again using `sqludf_substr()`.

2. Process most of an input LOB one part at a time.

   This UDF is looking for something in the LOB value. Most often it will find it near the front, but sometimes it may have to scan the entire 100 million byte value. The UDF uses `sqludf_length()` to find the size of this particular value, and steps through the value 1000 bytes at a time by placing a call to `sqludf_substr()` in a loop. It uses a variable as the starting position, increasing the variable by 1000 each time through the loop. It proceeds in this manner until it finds what it is looking for.

3. Return one of the two input LOBs

   This UDF has two LOB locators as inputs, and returns a LOB locator as an output. It examines and compares the two inputs, reading the bytes received using `sqludf_substr()` and then determines which of the two to select based on some algorithm. When it determines this, it copies the locator of the selected input to the buffer indicated by the UDF result argument, and exits.

4. Cut and paste an input LOB, and return the result.

   The UDF is passed a LOB value and maybe some other arguments which presumably tell it how to proceed. It creates a locator for its output, and proceeds to build the output value sequentially, taking most of the result value from different parts of the input LOB which it reads using `sqludf_substr()`, based on the instructions contained in the other input arguments. Finally when it is done it copies the result locator to the buffer to which the UDF result argument points, and then exits.

Other Coding Considerations

This section documents additional considerations for implementing a UDF, items to keep in mind, and items to avoid.

Hints and Tips

The following are recommendations to consider to successfully implement your UDF:

1. **UDF bodies need to be protected.** The executable function bodies are not captured or protected in any way by DB2. The CREATE FUNCTION statement merely points to the body. To preserve the integrity of the function and the database applications which depend on the function, you must, by managing access to the directory containing the function and by protecting the body itself, prevent the function body from being inadvertently or intentionally deleted or replaced.

2. DB2 passes pointers to all of the buffers in the interface between DB2 and SQL (that is, all the SQL arguments and the function return value). Be sure you define your UDF arguments as pointers.

3. All SQL argument values are buffered. This means that a copy of the value is made and presented to the UDF. If a UDF changes its input parameters, the changes will have no effect when the UDF returns to DB2.

4. For OLE automation, do not change the input parameters, otherwise memory resources may not be freed and you may encounter a memory leak.

   In case of a major OLE library version mismatch or a failure in initializing the OLE library, the database manager returns SQLCODE -465 (SQLSTATE 58032) with reason code 21, (`Failure to initialize OLE library`).

5. Reentrancy is **strongly recommended** for UDFs on all operating platforms, so that one copy of it can be used for multiple concurrent statements and applications.

   Note that the SCRATCHPAD facility can be used to circumvent many of the limitations imposed by re-entrancy.

6. If the body of a function currently being used is modified (for example, recompiled and relinked), DB2 will not *change functions* in mid-transaction. However, the copy used in a subsequent transaction may be different if this kind of dynamic modification is taking place. This practice is not recommended.

7. If you allocate dynamic memory in the UDF, it should be freed before returning to DB2. This is especially important for the NOT FENCED case. The SCRATCHPAD facility can be used, however, to anchor dynamic memory needed by the UDF across invocations. If you use the scratchpad in this manner, specify the FINAL CALL attribute on the CREATE FUNCTION for the UDF so that it can free the allocated memory at end-of-statement processing. The reason for this is that the system could run out of memory over time, with repeated use of the UDF.

   This reasoning holds as well for other system resources used by the UDF.

8. Use the NOT NULL CALL option if it makes sense to do so. With this CREATE FUNCTION option, you do not have to check whether each SQL argument is null, and it performs better when you do have NULL values.

9. Use the NOT DETERMINISTIC option if the result from your UDF depends on anything other than the input SQL arguments. This option prevents the SQL compiler from performing certain optimizations which can cause inconsistent results.

10. Use the EXTERNAL ACTION option if your UDF has any side effects which need to be reliably performed. EXTERNAL ACTION prevents the SQL compiler from performing certain optimizations which can prevent invocation of your UDF in certain circumstances.

11. Use the FENCED option, unless you are either:

    - Working on a test database under circumstances where the integrity of the database does not matter
    - Absolutely certain that the UDF can perform no potentially damaging actions resulting in an erroneous modification of storage. *DB2 provides some protection against these kinds of actions, but does not guarantee the integrity of the database if you run NOT FENCED*. See “Debugging your UDF” on page 347 for more information on running FENCED.

12. For considerations on using UDFs with EUC code sets, see “Considerations for UDFs” on page 380.

13. For an application running unfenced UDFs, the first time such a UDF is invoked, a block of memory of the size indicated by the `UDF_MEM_SZ` configuration parameter is created. Thereafter, on a statement by statement basis, memory for interfacing between DB2 and unfenced UDFs is allocated and deallocated from this block of memory as needed.

    For fenced UDFs, a different block of memory is used in the same way. It is different because the memory is shared between processes. In fact, if an application uses both unfenced and fenced UDFs, two separate blocks of memory, each of the size indicated by the `UDF_MEM_SZ` parameter are used. See the *Administration Guide* for more information about this configuration parameter.

14. Use the DISALLOW PARALLELISM option in the following situations:

    - On scalar UDFs, if your UDF absolutely depends on running the same copy. Generally, this will be the case for NOT DETERMINISTIC SCRATCHPAD UDFs. (For an example, see the COUNTER UDF specified in “Scratchpad Considerations” on page 312.)
    - If you do not want the UDF to run on multiple partitions at once for a single reference.
    - If you are specifying a table function.

    Otherwise, ALLOW PARALLELISM (the default) should be specified.

UDF Restrictions and Caveats

This section discusses items to be avoided in your UDF:

1. In general DB2 does not restrict the use of operating system functions. A few exceptions are:

    a. Registering of signal or exception handlers may interfere with DB2’s use of these same handlers and may result in unexpected failure.

    b. System calls that terminate a process may abnormally terminate one of DB2’s processes and result in system or application failure.

2. The values of all environment variables beginning with 'DB2' are captured at the time the database manager is started with db2start, and are available in all UDFs whether or not they are fenced. The only exception is the DB2CKPTR environment variable. Note that the environment variables are *captured*; any changes to the environment variables after db2start is issued are not available to the UDFs.

3. With respect to LOBs passed to an external UDF, you are limited to the maximum size specified by the *UDF Shared Memory Size* DB2 system configuration parameter. The maximum that you can specify for this parameter is 256M. The default setting on DB2 is 1M. For more information on this parameter, see the *Administration Guide*.

4. Input to, and output from, the screen and keyboard is not recommended. In the process model of DB2, UDFs run in the background, so you cannot write to the screen. However, you can write to a file.

   Note: DB2 does not attempt to synchronize any external input/output performed by a UDF with DB2's own transactions. So for example, if a UDF writes to a file during a transaction, and that transaction is later backed out for some reason, no attempt is made to discover or undo the writes to the file.

5. On UNIX-based systems, your UDF runs under the UID of the DB2 Agent Process (NOT FENCED), or the UID which owns the db2udf executable (FENCED). This UID controls the system resources available to the UDF. For information on the db2udf executable, refer to the *Quick Beginnings*.

6. When using protected resources, (that is, resources that only allow one process access at a time) inside UDFs, you should try to avoid deadlocks between UDFs. If two or more UDFs deadlock, DB2 will not be able to detect the condition.

7. Character data is passed to external functions in the code page of the database. Likewise, a character string that is output from the function is assumed by the database to use the database's code page. In the case where the application code page differs from the database code page, the code page conversions occur as they would for other values in the SQL statement. You can prevent this conversion, by coding FOR BIT DATA as an attribute of the character parameter or result in your CREATE FUNCTION statement. If the character parameter is not defined with the FOR BIT DATA attribute, your UDF code will receive arguments in the database code page.

   Note that, using the DBINFO option on CREATE FUNCTION, the database code page is passed to the UDF. Using this information, a UDF which is sensitive to the code page can be written to operate in many different code pages.

8. When writing a UDF using C++, you may want to consider declaring the function name as:

   ```
   extern "C" void SQL_API_FN udf( ...arguments... )
   ```

   The extern "C" prevents type decoration (or 'mangling') of the function name by the C++ compiler. Without this declaration, you have to include all the type decoration for the function name when you issue the CREATE FUNCTION statement.

Examples of UDF Code

The following UDF code examples are supplied with DB2.

Example: Integer Divide Operator
Example: Fold the CLOB, Find the Vowel
Example: Counter

For information on where to find all the examples supplied, and how to invoke them, see“Installing, Building and Executing the Sample Programs” on page 527.

For information on compiling and linking UDFs, refer to the *DB2 SDK Building Applications* book for your platform.

Each of the example UDFs is accompanied by the corresponding CREATE FUNCTION statement, and a small scenario showing its use. These scenarios all use the following table TEST, which has been carefully crafted to illustrate certain points being made in the scenarios. Here is the table definition:

```
CREATE TABLE TEST (INT1 INTEGER,
                   INT2 INTEGER,
                   PART CHAR(5),
                   DESCR CLOB(33K))
```

After population of the table, the following statement was issued using CLP to display its contents:

```
SELECT INT1, INT2, PART, SUBSTR(DESCR,1,50) FROM TEST
```

Note the use of the SUBSTR function on the CLOB column to make the output more readable. This resulted in the following CLP output:

```
INT1        INT2        PART  4
----------- ----------- ----- --------------------------------------------------
         16           1 brain The only part of the body capable of forgetting.
          8           2 heart The seat of the emotions?
          4           4 elbow That bendy place in mid-arm.
          2           0 -     -
         97          16 xxxxx Unknown.
  5 record(s) selected.
```

The above information on table TEST is for your reference as you read the examples and scenarios which follow.

Example: Integer Divide Operator

Suppose you are unhappy with the way integer divide works in DB2, because it returns an error, SQLCODE -802 (SQLSTATE 22003), and terminates the statement when the divisor is zero. Instead, you want the integer divide to return a NULL, so you code this UDF:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

 /*************************************************************************
*  function divid: performs integer divid, but unlike the / operator
*                  shipped with the product, gives NULL when the
*                  denominator is zero.
*
*                This function does not use the constructs defined in the
*                "sqludf.h" header file.
*
*     inputs:  INTEGER num     numerator
*              INTEGER denom   denominator
*     output:  INTEGER out     answer
**************************************************************************/
void SQL_API_FN divid (
   long *num,                           /* numerator */
   long *denom,                         /* denominator */
   long *out,                           /* output result */
   short *in1null,                      /* input 1 NULL indicator */
   short *in2null,                      /* input 2 NULL indicator */
   short *outnull,                      /* output NULL indicator */
   char *sqlstate,                      /* SQL STATE */
   char *funcname,                      /* function name */
   char *specname,                      /* specific function name */
   char *mesgtext) {                    /* message text insert */

   if (*denom == 0) {      /* if denominator is zero, return null result */
      *outnull = -1;
   } else {                /* else, compute the answer */
      *out = *num / *denom;
      *outnull = 0;
   } /* endif */
   return;
}
/* end of UDF : divid */
```

For this UDF, notice that:

- It does not include `sqludf.h`.
- It has two input arguments defined, and one output argument.
- It is defined to return `void`. Remember that the normal UDF outputs will be returned using the input arguments.
- The inclusion of SQL_API_FN in the function definition is designed to assure portability of function source across platforms. It requires the inclusion of the following statement in your UDF source files.

  ```
  #include <sqlsystm.h>
  ```

- It does not check for null input arguments, because the NOT NULL CALL parameter is specified by default in the CREATE FUNCTION statement shown below.

Here is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION MATH."/"(INT,INT)
  RETURNS INT
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME '/u/slick/udfx/div' ;
```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)

For this statement, observe that:

- It is defined to be in the MATH schema. In order to define a UDF in a schema that is not equal to your user-ID, you need DBADM authority on the database.
- The function name is defined to be "/", the same name as the SQL divide operator. In fact, this UDF can be invoked the same as the built-in / operator, using either *infix notation*, for example, A / B, or *functional notation*, for example, "/"(A,B). See below.
- You have chosen to define it as NOT FENCED because you are absolutely sure that the program has no errors.
- You have used the default NOT NULL CALL, by which DB2 provides a NULL result if either argument is NULL, without invoking the body of the function.

Now if you run the following pair of statements (CLP input is shown):

```
SET CURRENT FUNCTION PATH = SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

You get this output from CLP:

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
SQL0802N  Arithmetic overflow or other arithmetic exception occurred.
SQLSTATE=22003
```

The SQL0802N error message occurs because you have set your CURRENT FUNCTION PATH special register to a concatenation of schemas which does not include MATH, the schema in which the "/" UDF is defined. And therefore you are executing DB2's built-in divide operator, whose defined behavior is to give the error when a "divide by zero" condition occurs. The fourth row in the TEST table provides this condition.

However, if you change the function path, putting MATH in front of SYSIBM in the path, and rerun the SELECT statement:

```
SET CURRENT FUNCTION PATH = MATH, SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

You then get the desired behavior, as shown by the following CLP output:

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
          2           0           -           -
         97          16           6           6
  5 record(s) selected.
```

For the above example, observe that:

- The SET CURRENT FUNCTION PATH statement changes the current function path used in the following statement as this is dynamic SQL, placing the MATH schema ahead of SYSIBM.
- The fourth row produces a NULL result from the divides, and the statement continues.
- Both syntaxes (*infix syntax*, and *prefix syntax*) can be used to invoke this particular UDF, because its name is the same as a built-in operator, and both *are used* in the above example, with identical results.
- As a practical note, because of the way the built-in functions and operators are defined to DB2, this "/" is not used for operations on SMALLINTs. The DB2 function selection algorithm chooses the exact match built-in "/" operator in preference to this user-defined "/", which is a match but not an exact match. There are different ways around this seeming inconsistency. You can explicitly cast SMALLINT arguments to INTEGER before invoking "/", for example, `INT1 / INTEGER(SMINT1)` (where the column SMINT1 is assumed to be SMALLINT). Or, better than that, you could register additional UDFs, further overloading the "/" operator, which define first and second parameters that are SMALLINT. These additional UDFs can be sourced on `MATH."/"`.

  In this case, for a fully general set of functions you have to CREATE the following three additional functions to completely handle integer divide:

  ```
  CREATE FUNCTION MATH."/"(SMALLINT,SMALLINT)
    RETURNS INT
    SOURCE MATH."/"(INT,INT)

  CREATE FUNCTION MATH."/"(SMALLINT,INT)
    RETURNS INT
  ```

```
    SOURCE MATH."/"(INT,INT)

  CREATE FUNCTION MATH."/"(INT,SMALLINT)
    RETURNS INT
    SOURCE MATH."/"(INT,INT)
```

Even though three UDFs are added, additional code does not have to be written as they are sourced on `MATH."/"`.

And now, with the definition of these four "/" functions, any users who want to take advantage of the new behavior on integer divide need only place MATH ahead of SYSIBM in their function path, and can write their SQL as usual.

Example: Fold the CLOB, Find the Vowel

Suppose you have coded up two UDFs to help you with your text handling application. The first UDF *folds* your text string after the *n*th byte. In this example, fold means to put the part that was originally after the *n* byte before the part that was originally in front of the *n+1* byte. In other words, the UDF moves the first n bytes from the beginning of the string to the end of the string. The second function returns the position of the first vowel in the text string. Both of these functions are coded in the `udf.c` example file:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

 /*************************************************************************
*  function fold: input string is folded at the point indicated by the
*                 second argument.
*
*     input: CLOB    in1         input string
*            INTEGER in2         position to fold on
*            CLOB    out         folded string
*************************************************************************/
void SQL_API_FN fold (
   SQLUDF_CLOB     *in1,               /* input CLOB to fold */
   SQLUDF_INTEGER  *in2,               /* position to fold on */
   SQLUDF_CLOB     *out,               /* output CLOB, folded */
   SQLUDF_NULLIND  *in1null,           /* input 1 NULL indicator */
   SQLUDF_NULLIND  *in2null,           /* input 2 NULL indicator */
   SQLUDF_NULLIND  *outnull,           /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                /* trailing arguments */

   SQLUDF_INTEGER len1;

   if (SQLUDF_NULL(in1null) || SQLUDF_NULL(in2null)) {
      /* one of the arguments is NULL.  The result is then "INVALID INPUT" */
      strcpy (out->data, "INVALID INPUT");
      out->length = strlen("INVALID INPUT");
   } else {
      len1 = in1->length;              /* length of the CLOB */

      /* build the output by folding at position "in2" */
      strncpy (out->data, &in1->data[*in2], len1 - *in2);
      strncpy (&out->data[len1 - *in2], in1->data, *in2);
      out->length = in1->length;
   } /* endif */
   *outnull = 0;                        /* result is always non-NULL */
   return;
}
/* end of UDF : fold */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

 /*************************************************************************
*  function findvwl: returns the position of the first vowel.
*                 returns an error if no vowel is found
*                 when the function is created, must be defined as
*                 NOT NULL CALL.
*     inputs: VARCHAR(500) in
*     output: INTEGER      out
**************************************************************************/
void SQL_API_FN findvwl (
   SQLUDF_VARCHAR   *in,                /* input character string */
   SQLUDF_SMALLINT  *out,               /* output location of vowel */
   SQLUDF_NULLIND   *innull,            /* input NULL indicator */
   SQLUDF_NULLIND   *outnull,           /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                 /* trailing arguments */

   short i;                             /* local indexing variable */

   for (i=0; (i < (short)strlen(in) &&         /* find the first vowel */
      in[i] != 'a' && in[i] != 'e' && in[i] != 'i' &&
      in[i] != 'o' && in[i] != 'u' && in[i] != 'y' &&
      in[i] != 'A' && in[i] != 'E' && in[i] != 'I' &&
      in[i] != 'O' && in[i] != 'U' && in[i] != 'Y'); i++);
   if (i == strlen(in)) {               /* no vowels found */
      strcpy (sqludf_sqlstate, "38999");               /* error state */
      strcpy (sqludf_msgtext, "findvwl: No Vowel");    /* message insert */
   } else {                             /* a vowel was found at "i" */
      *out = i + 1;
      *outnull = 0;
   } /* endif */
   return;
}
/* end of UDF : findvwl */
```

For the above UDFs, notice:

- They include `sqludf.h`, and use the argument definitions and macros contained in that file.
- The `fold()` function is invoked even with NULL arguments, and returns the string `INVALID INPUT` in this case. The `findvwl()` function on the other hand, is not

invoked with null arguments. The use of the SQLUDF_NULL() macro, defined in sqludf.h checks for null arguments in fold().

- The findvwl() function sets the error SQLSTATE and the message token.
- The fold() function returns a CLOB value in addition to having the CLOB data type as its text input argument. The findvwl() has a VARCHAR input argument.

Here are the CREATE FUNCTION statements for these UDFs:

```
CREATE FUNCTION FOLD(CLOB(100K),INT)
  RETURNS CLOB(100K)
  FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!fold' ;

CREATE FUNCTION FINDV(VARCHAR(500))
  RETURNS INTEGER
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NOT NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!findvwl' ;
```

The above CREATE FUNCTION statements are for UNIX-based platforms. On other platforms, you may need to modify the value specified in the EXTERNAL NAME clause in the above statements. You can find the above CREATE FUNCTION statements in the calludf.sqc example program shipped with DB2.

Referring to these CREATE statements, observe that:

- The schema names of the functions will default to the statement authorization-ID.
- You have chosen to define FOLD as FENCED because you are not absolutely sure that it is error-free, but FINDV is NOT FENCED.
- You have coded NULL CALL for FOLD, which means that fold() will be called even if either input argument is null, which agrees with the way the function is coded. FINDV is coded to be NOT NULL CALL, which also agrees with the code.
- Both will default to ALLOW PARALLELISM.

Now you can successfully run the following statement:

```
SELECT SUBSTR(DESCR,1,30), SUBSTR(FOLD(DESCR,6),1,30) FROM TEST
```

The output from the CLP for this statement is:

```
1                              2
------------------------------ ------------------------------
The only part of the body capa ly part of the body capable of
The seat of the emotions?      at of the emotions?The se
That bendy place in mid-arm.   endy place in mid-arm.That b
-                              INVALID INPUT
Unknown.                       n.Unknow
  5 record(s) selected.
```

Note the use of the SUBSTR built-in function to make the selected CLOB values display more nicely. It shows how the output is folded (best seen in the second, third and fifth rows, which have a shorter CLOB value than the first row, and thus the folding is more evident even with the use of SUBSTR). And it shows (fourth row) how the INVALID INPUT string is returned by the FOLD UDF when its input text string (column DESCR) is null. This SELECT also shows simple nesting of function references; the reference to FOLD is within an argument of the SUBSTR function reference.

Then if you run the following statement:

```
SELECT PART, FINDV(PART) FROM TEST
```

The CLP output is as follows:

```
PART  2
----- -----------
brain           3
heart           2
elbow           1
-               -
SQL0443N  User defined function "SLICK.FINDV" (specific name
"SQL950424135144750") has returned an error SQLSTATE with diagnostic
text "findvwl: No Vowel".  SQLSTATE=38999
```

This example shows how the 38999 SQLSTATE value and error message token returned by findvwl() are handled: message SQL0443N returns this information to the user. The PART column in the fifth row contains no vowel, and this is the condition which triggers the error in the UDF.

Observe the argument promotion in this example. The PART column is CHAR(5), and is promoted to VARCHAR to be passed to FINDV.

And finally note how DB2 has generated a null output from FINDV for the fourth row, as a result of the NOT NULL CALL specification in the CREATE statement for FINDV.

The following statement:

```
SELECT SUBSTR(DESCR,1,25), FINDV(CAST (DESCR AS VARCHAR(60) ) )
FROM TEST
```

Produces this output when executed in the CLP:

```
1                        2
------------------------ -----------
The only part of the body          3
The seat of the emotions?          3
That bendy place in mid-a          3
-                                  -
Unknown.                           1
  5 record(s) selected.
```

This SELECT statement shows FINDV working on a VARCHAR input argument. Observe how we cast column DESCR to VARCHAR to make this happen. Without the cast we would not be able to use FINDV on a CLOB argument, because CLOB is not promotable to VARCHAR. Again, the built-in SUBSTR function is used to make the DESCR column value display better.

And here again note that the fourth row produces a null result from FINDV because of the NOT NULL CALL.

Example: Counter

Suppose you want to simply number the rows in your SELECT statement. So you write a UDF which increments and returns a counter. This UDF uses a scratchpad:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
   long len;
   long countr;
   char not_used[96];
};

 /*************************************************************************
*  function ctr: increments and reports the value from the scratchpad.
*
*                This function does not use the constructs defined in the
*                "sqludf.h" header file.
*
*     input:  NONE
*     output: INTEGER out      the value from the scratchpad
**************************************************************************/
void SQL_API_FN ctr (
   long *out,                          /* output answer (counter) */
   short *outnull,                     /* output NULL indicator */
   char *sqlstate,                     /* SQL STATE */
   char *funcname,                     /* function name */
   char *specname,                     /* specific function name */
   char *mesgtext,                     /* message text insert */
   struct scr *scratchptr) {           /* scratch pad */
   *out = ++scratchptr->countr;      /* increment counter & copy out */
   *outnull = 0;
   return;
}
/* end of UDF : ctr */
```

For this UDF, observe that:

- It does not include `sqludf.h`. But it does include `sqlsystm.h` for the definition of SQL_API_FN.
- It has no input SQL arguments defined, but returns a value.
- It appends the scratchpad input argument after the four standard trailing arguments, namely *SQL-state*, *function-name*, *specific-name*, and *message-text*.
- It includes a structure definition to map the scratchpad which is passed.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  NOT FENCED
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!ctr'
  DISALLOW PARALLELISM;
```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)

Referring to this statement, observe that:

- No input parameters are defined. This agrees with the code.
- SCRATCHPAD is coded, causing DB2 to allocate, properly initialize and pass the scratchpad argument.
- You have chosen to define it as NOT FENCED because you are absolutely sure that it is error free.
- You have specified it to be NOT DETERMINISTIC, because it depends on more than the SQL input arguments, (none in this case).
- You have correctly specified DISALLOW PARALLELISM, because correct functioning of the UDF depends on a single scratchpad.

Now you can successfully run the following statement:

```
SELECT INT1, COUNTER(), INT1/COUNTER() FROM TEST
```

When run through the CLP, it produces this output:

```
INT1        2           3
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19
  5 record(s) selected.
```

Observe that the second column shows the straight COUNTER() output. The third column shows that the two separate references to COUNTER() in the SELECT statement each get their own scratchpad; had they not each gotten their own, the output in the second column would have been `1 3 5 7 9`, instead of the nice orderly `1 2 3 4 5`.

Example: Weather Table Function

The following is an example table function, `tfweather_u`, (supplied by DB2 in the programming example `tblsrv.c`), that returns weather information for various cities in the United States. The weather data for these cities is included in the example program, but could be read in from an external file, as indicated in the comments contained in the example program. The data includes the name of a city followed by its weather information. This pattern is repeated for the other cities. Note that there is a client application (`tblcli.sqc`) supplied with DB2 that calls this table function and prints out the weather data retrieved using the `tfweather_u` table function.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h> /* for use in compiling User Defined Function */
#include "util.h"

#define   SQL_NOTNULL   0   /* Nulls Allowed - Value is not Null */
#define   SQL_ISNULL   -1   /* Nulls Allowed - Value is Null */

/* Short and long city name structure */
typedef struct {
  char * city_short ;
  char * city_long ;
} city_area ;

/* Scratchpad data */
/* Preserve information from one function call to the next call */
typedef struct {
  /* FILE * file_ptr; if you use weather data text file */
  int file_pos ;  /* if you use a weather data buffer */
} scratch_area ;

/* Field type conversion (text to SQL) */
typedef enum {
  udf_varchar,
  udf_integer,
  udf_double
  /* You may want to add more field types here */
} udf_types ;

/* Field descriptor structure */
typedef struct {
  char      fld_field[31] ;                     /* Field data */
  int       fld_ind ;            /* Field null indicator data */
  udf_types fld_type ;                          /* Field type */
  int       fld_length ;  /* Field length in the weather data */
  int       fld_offset ;  /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
  { "alb", "Albany, NY"                  },
  { "atl", "Atlanta, GA"                 },
  .
  .
  .
  { "wbc", "Washington DC, DC"           },
  /* You may want to add more cities here */

  /* Do not forget a null termination */
  { ( char * ) 0, ( char * ) 0           },
} ;

/* Field descriptor data */
fld_desc fields[] = {
  { "", SQL_ISNULL, udf_varchar, 30,  0 }, /* city          */
  { "", SQL_ISNULL, udf_integer,  3,  2 }, /* temp_in_f     */
  { "", SQL_ISNULL, udf_integer,  3,  7 }, /* humidity      */
  { "", SQL_ISNULL, udf_varchar,  5, 13 }, /* wind          */
  { "", SQL_ISNULL, udf_integer,  3, 19 }, /* wind_velocity */
  { "", SQL_ISNULL, udf_double,   5, 24 }, /* barometer     */
  { "", SQL_ISNULL, udf_varchar, 25, 30 }, /* forecast      */
  /* You may want to add more fields here */

  /* Do not forget a null termination */
  { ( char ) 0, 0, ( udf_types ) 0, 0, 0 },
} ;

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement.  Note that you  */
/* have to specify the full path name for this file.         */
char * weather_data[] = {
   "alb.forecast",
   "   34   28%    wnw   3  30.53 clear",
   "atl.forecast",
   "   46   89%   east  11  30.03 fog",
   "aus.forecast",
   "   59   57%  south   5  30.05 clear",
   .
   .
   .
   "wbc.forecast",
   "   38   96%    ene  16  30.31 light rain",
   /* You may want to add more weather data here */

   /* Do not forget a null termination */
   ( char * ) 0,
} ;

/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
       if (strcmp(short_name, cities[name_pos].city_short) == 0) {
          strcpy( long_name, cities[name_pos].city_long ) ;
          /* A full city name found */
          return( 0 ) ;
       }
       name_pos++ ;
    }
    /* Could not find such city in the city data */
    strcpy( long_name, "Unknown City" ) ;
    return( -1 ) ;

}

/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {

    while ( fields[field_pos].fld_length != 0 ) {
       memset( fields[field_pos].fld_field, '\0', 31 ) ;
       fields[field_pos].fld_ind = SQL_ISNULL ;
       field_pos++ ;
    }
    return( 0 ) ;

}

/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

    fld_desc * field ;
    char field_buf[31] ;
    double * double_ptr ;
    int * int_ptr, buf_pos ;

    while ( fields[field_pos].fld_length != 0 ) {
       field = &fields[field_pos] ;
       memset( field_buf, '\0', 31 ) ;
       memcpy( field_buf,
               ( value + field->fld_offset ),
               field->fld_length ) ;
       buf_pos = field->fld_length ;
       while ( ( buf_pos > 0 ) &&
               ( field_buf[buf_pos] == ' ' ) )
          field_buf[buf_pos--] = '\0' ;
       buf_pos = 0 ;
       while ( ( buf_pos < field->fld_length ) &&
               ( field_buf[buf_pos] == ' ' ) )
          buf_pos++ ;
       if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
            strcmp( ( char * ) ( field_buf + buf_pos ), "n/a") != 0 ) {
          field->fld_ind = SQL_NOTNULL ;

          /* Text to SQL type conversion */
          switch( field->fld_type ) {
            case udf_varchar:
                 strcpy( field->fld_field,
                         ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case udf_integer:
                 int_ptr = ( int * ) field->fld_field ;
                 *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case udf_double:
                 double_ptr = ( double * ) field->fld_field ;
                 *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            /* You may want to add more text to SQL type conversion here */
          }

       }
       field_pos++ ;
    }
    return( 0 ) ;

}

void weather( /* Return row fields */
              SQLUDF_VARCHAR * city,
              SQLUDF_INTEGER * temp_in_f,
              SQLUDF_INTEGER * humidity,
              SQLUDF_VARCHAR * wind,
              SQLUDF_INTEGER * wind_velocity,
              SQLUDF_DOUBLE  * barometer,
              SQLUDF_VARCHAR * forecast,
              /* You may want to add more fields here */

              /* Return row field null indicators */
              SQLUDF_NULLIND * city_ind,
              SQLUDF_NULLIND * temp_in_f_ind,
              SQLUDF_NULLIND * humidity_ind,
              SQLUDF_NULLIND * wind_ind,
              SQLUDF_NULLIND * wind_velocity_ind,
              SQLUDF_NULLIND * barometer_ind,
              SQLUDF_NULLIND * forecast_ind,
              /* You may want to add more field indicators here */

              /* UDF always-present (trailing) input arguments */
              SQLUDF_TRAIL_ARGS_ALL
            ) {

  scratch_area * save_area ;
  char line_buf[81] ;
  int line_buf_pos, unknown_city ;

  /* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
  /* Preserve information from one function call to the next call */
  save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

  /* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
  switch( SQLUDF_CALLT ) {

    /* First call UDF: Open table and fetch first row */
    case SQL_TF_OPEN:
         /* If you use a weather data text file specify full path */
         /* save_area->file_ptr = fopen("/sqllib/samples/c/tblsrv.dat",
                                        "r"); */
         save_area->file_pos = 0 ;
         break ;

    /* Normal call UDF: Fetch next row */
    case SQL_TF_FETCH:
         /* If you use a weather data text file */
         /* memset(line_buf, '\0', 81); */
         /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
         if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

            /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
            strcpy( SQLUDF_STATE, "02000" ) ;

            break ;
         }
         memset( line_buf, '\0', 81 ) ;
         strcpy( line_buf, weather_data[save_area->file_pos] ) ;
         line_buf[3] = '\0' ;
         unknown_city = 0 ;

         /* Clean all field data and field null indicator data */
         clean_fields( 0 ) ;
```

```
/* Find a full city name using a short name */
/* Fills city field data */
if ( get_name( line_buf, fields[0].fld_field ) < 0 )
   unknown_city = 1 ;

/* Fills city field null indicator data */
fields[0].fld_ind = SQL_NOTNULL ;

if ( !unknown_city ) {
   save_area->file_pos++ ;
   /* If you use a weather data text file */
   /* memset(line_buf, '\0', 81); */
   /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
   if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

      /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
      strcpy( SQLUDF_STATE, "02000" ) ;

      break ;
   }
   memset( line_buf, '\0', 81 ) ;
   strcpy( line_buf, weather_data[save_area->file_pos] ) ;
   line_buf_pos = strlen( line_buf ) ;
   while ( line_buf_pos > 0 ) {
      if ( line_buf[line_buf_pos] >= ' ' )
         line_buf_pos = 0 ;
      else {
         line_buf[line_buf_pos] = '\0' ;
         line_buf_pos-- ;
      }
   }

   /* Fills field data and field null indicator data ... */
   /* ... for selected city from text weather data */
   get_value( line_buf, 1 ) ;  /* Skips city field */

   /* Builds return row fields */
   strcpy( city, fields[0].fld_field ) ;
   memcpy( (void *) temp_in_f,
           fields[1].fld_field,
           sizeof( SQLUDF_INTEGER ) ) ;
   memcpy( (void *) humidity,
           fields[2].fld_field,
           sizeof( SQLUDF_INTEGER ) ) ;
   strcpy( wind, fields[3].fld_field ) ;
   memcpy( (void *) wind_velocity,
           fields[4].fld_field,
           sizeof( SQLUDF_INTEGER ) ) ;
   memcpy( (void *) barometer,
           fields[5].fld_field,
           sizeof( SQLUDF_DOUBLE ) ) ;
   strcpy( forecast, fields[6].fld_field ) ;

   /* Builds return row field null indicators */
   memcpy( (void *) city_ind,
           &(fields[0].fld_ind),
           sizeof( SQLUDF_NULLIND ) ) ;
   memcpy( (void *) temp_in_f_ind,
           &(fields[1].fld_ind),
           sizeof( SQLUDF_NULLIND ) ) ;
   memcpy( (void *) humidity_ind,
           &(fields[2].fld_ind),
           sizeof( SQLUDF_NULLIND ) ) ;
   memcpy( (void *) wind_ind,
           &(fields[3].fld_ind),
           sizeof( SQLUDF_NULLIND ) ) ;
   memcpy( (void *) wind_velocity_ind,
                   &(fields[4].fld_ind),
                   sizeof( SQLUDF_NULLIND ) ) ;
           memcpy( (void *) barometer_ind,
                   &(fields[5].fld_ind),
                   sizeof( SQLUDF_NULLIND ) ) ;
           memcpy( (void *) forecast_ind,
                   &(fields[6].fld_ind),
                   sizeof( SQLUDF_NULLIND ) ) ;

        }

        /* Next city weather data */
        save_area->file_pos++ ;

        break ;

   /* Special last call UDF for cleanup (no real args!): Close table */
   case SQL_TF_CLOSE:
        /* If you use a weather data text file */
        /* fclose(save_area->file_ptr); */
        /* save_area->file_ptr = NULL; */
        save_area->file_pos = 0 ;
        break ;

  }

}
```

Referring to this UDF code, observe that:

- The scratchpad is defined. The `row` variable is initialized on the OPEN call, and the `iptr` array and `nbr_rows` variable are filled in by the *mystery* function at OPEN time.
- FETCH traverses the `iptr` array, using `row` as an index, and moves the values of interest from the current element of `iptr` to the location pointed to by `out_c1`, `out_c2`, and `out_c3` result value pointers.
- Finally CLOSE frees the storage acquired by OPEN and anchored in the scratchpad.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION tfweather_u()
  RETURNS TABLE (CITY VARCHAR(25),
                 TEMP_IN_F INTEGER,
                 HUMIDITY INTEGER,
                 WIND VARCHAR(5),
                 WIND_VELOCITY INTEGER,
                 BAROMETER FLOAT,
                 FORECAST VARCHAR(25))
  SPECIFIC tfweather_u
  DISALLOW PARALLELISM
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  FINAL CALL
```

```
        LANGUAGE C
        PARAMETER STYLE DB2SQL
        EXTERNAL NAME 'tf_dml!weather';
```

The above CREATE FUNCTION statement is for a UNIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.

Referring to this statement, observe that:

- It does not take any input, and returns 7 output columns.
- SCRATCHPAD is specified, so DB2 allocates, properly initializes and passes the scratchpad argument.
- FINAL CALL is specified; this is mandatory for a table function.
- The function is specified as NOT DETERMINISTIC, because it depends on more than the SQL input arguments. That is, it depends on the `mystery` function and we assume that the content can vary from execution to execution.
- DISALLOW PARALLELISM is required for table functions.
- CARDINALITY 100 is an estimate of the expected number of rows returned, provided to the DB2 optimizer.
- DBINFO is not used, and the optimization to only return the columns needed by the particular statement referencing the function is not implemented.
- NOT NULL CALL is specified, so the UDF will not be called if any of its input SQL arguments are NULL, and does not have to check for this condition.

Example: Function using LOB locators

This UDF takes a locator for an input LOB, and returns a locator for another LOB which is a subset of the input LOB. There are some criteria passed as a second input value, which tell the UDF how exactly to break up the input LOB.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqludf.h>
#include "util.h"

void SQL_API_FN lob_subsetter(
                udf_locator * lob_input,   /* locator of LOB value to carve up */
                char  * criteria,          /* criteria for carving */
                udf_locator * lob_output,  /* locator of result LOB value */
                short * inp_nul,
                short * cri_nul,
                short * out_nul,
                char  * sqlstate,
                char  * funcname,
                char  * specname,
                char  * msgtext ) {
```

```
/* local vars */
short j;              /* local indexing var */
int   rc;             /* return code variable for API calls */
long  input_len;      /* receiver for input LOB length */
long  input_pos;      /* current position for scanning input LOB */
char lob_buf[100];    /* data buffer */
long  input_rec;      /* number of bytes read by sqludf_substr */
long  output_rec;     /* number of bytes written by sqludf_append */

/*---------------------------------------------
 * UDF Program Logic Starts Here
 *---------------------------------------------
 * What we do is create an output handle, and then
 * loop over the input, 100 bytes at a time.
 * Depending on the "criteria" passed in, we may decide
 * to append the 100 byte input lob segment to the output, or not.
 *---------------------------------------------
 * Create the output locator, right in the return buffer.
 */

rc = sqludf_create_locator(SQL_TYP_CLOB, &lob_output);
/* Error and exit if unable to create locator */
if (rc) {
   memcpy (sqlstate, "38901", 5); /* special sqlstate for this condition */
   goto exit;
}
/* Find out the size of the input LOB value */
rc = sqludf_length(lob_input, &input_len) ;
/* Error and exit if unable to find out length */
if (rc) {
   memcpy (sqlstate, "38902", 5); /* special sqlstate for this condition */
   goto exit;
}
/* Loop to read next 100 bytes, and append to result if it meets
 * the criteria.
 */
for (input_pos = 0; (input_pos < input_len); input_pos += 100) {
  /* Read the next 100 (or less) bytes of the input LOB value */
  rc = sqludf_substr(lob_input, input_pos, 100,
                     (unsigned char *) lob_buf, &input_rec) ;
  /* Error and exit if unable to read the segment */
  if (rc) {
     memcpy (sqlstate, "38903", 5); /* special sqlstate for this condition */
     goto exit;
  }
  /* apply the criteria for appending this segment to result
   * if (...predicate involving buffer and criteria...) {
   * The condition for retaining the segment is TRUE...
   * Write that buffer segment which was last read in
   */
  rc = sqludf_append(lob_output,
                     (unsigned char *) lob_buf, input_rec, &output_rec) ;
  /* Error and exit if unable to read the 100 byte segment */
  if (rc) {
     memcpy (sqlstate, "38904", 5); /* special sqlstate for this condition */
     goto exit;
  }
  /* } end if criteria for inclusion met */
} /* end of for loop, processing 100-byte chunks of input LOB
```

```
     * if we fall out of for loop, we are successful, and done.
         */
     *out_nul = 0;
  exit: /* used for errors, which will override null-ness of output. */
     return;
  }
```

Referring to this UDF code, observe that:

- There are includes for `sql.h`, where the type `SQL_TYP_CLOB` used in the `sqludf_create_locator()` call is defined, and `sqludf.h`, where the type `udf_locator` is defined.
- The first input argument, and the third input argument (which represents the function output) are defined as pointers to `sqludf_locator`, that is, they represent CREATE FUNCTION specifications of AS LOCATOR.
- There is no testing for nulls of either input argument as NOT NULL CALL is specified in the CREATE FUNCTION statement.
- In the event of error, the UDF exits with `sqlstate` set to 38*xxx*. This is sufficient to stop the execution of the statement referencing the UDF. The actual 38*xxx* SQLSTATE values you choose are not important to DB2, but can serve to differentiate the exception conditions which your UDF may encounter.
- The inclusion criteria are left unspecified, but in this case would presumably somehow determine if this particular buffer content passes the test, and presumably would account for the possibility that the last buffer might be a partial buffer.
- By using the `input_rec` variable as the length of the data appended, the UDF takes care of any partial buffer condition.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION carve(CLOB(50M), VARCHAR(255) )
  RETURNS CLOB(50M)
  NOT NULL CALL
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME '/u/wilfred/udfs/lobudfs!lob_subsetter' ;
```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)

Referring to this statement, observe that:

- NOT NULL CALL is specified, so the UDF will not be called if any of its input SQL arguments are NULL, and does not have to check for this condition.

- The function is defined to be NOT FENCED; recall that the APIs only work in NOT FENCED. NOT FENCED means that the definer will have to have the CREATE_NOT_FENCED authority on the database (which is also implied by DBADM authority).
- The function is specified as DETERMINISTIC, meaning that with a given input CLOB value and a given set of criteria, the result will be the same every time.

Now you can successfully run the following statement:

```
UPDATE tablex
  SET col_a = 99,
      col_b = carve (:hv_clob, '...criteria...')
  WHERE tablex_key = :hv_key;
```

The UDF is used to subset the CLOB value represented by the host variable :hv_clob and update the row represented by key value in host variable :hv_key.

In this update example by the way, it may be that :hv_clob is defined in the application as a CLOB_LOCATOR. It is not this same locator which will be passed to the "carve" UDF! When :hv_clob is "bound in" to the DB2 engine agent running the statement, it is known only as a CLOB. When it is then passed to the UDF, DB2 generates a new locator for the value. This conversion back and forth between CLOB and locator is not expensive, by the way; it does not involve any extra memory copies or I/O.

Example: Counter OLE Automation UDF in BASIC

The following example implements a counter class using Microsoft Visual BASIC. The class has an instance variable, `nbrOfInvoke`, that tracks the number of invocations. The constructor of the class initializes the number to 0. The increment method increments `nbrOfInvoke` by 1 and returns the current state.

```
Description="Example in SQL Reference"
Name="bert"
Class=bcounter; bcounter.cls
ExeName32="bert_app.exe"


VERSION 1.0 CLASS
BEGIN
  SingleUse = -1  'True
END
Attribute VB_Name = "bcounter"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit
Dim nbrOfInvoke As Long

Public Sub increment(output As Long, _
                     output_ind As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                     msg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

```
    nbrOfInvoke = nbrOfInvoke + 1

End Sub

Private Sub Class_Initialize()
  nbrOfInvoke = 0
End Sub

Private Sub Class_Terminate()

End Sub
```

The `bcounter` class is implemented as an OLE automation object and registered under the progId `bert.bcounter`. You can compile the automation server either as an in-process or local server; this is transparent to DB2. The following CREATE FUNCTION statement registers a UDF `bcounter` with the `increment` method as an external implementation:

```
CREATE FUNCTION bcounter () RETURNS integer
   EXTERNAL NAME 'bert.bcounter!increment'
   LANGUAGE OLE
   FENCED
   SCRATCHPAD
   FINAL CALL
   NOT DETERMINISTIC
   NULL CALL
   PARAMETER STYLE DB2SQL
   NO SQL
   NO EXTERNAL ACTION
   DISALLOW PARALLEL;
```

For the following query:

```
SELECT INT1, BCOUNTER() AS COUNT, INT1/BCOUNTER() AS DIV FROM TEST
```

The results are exactly the same as in the previous example:

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19

  5 record(s) selected.
```

Example: Counter OLE Automation UDF in C++

The following example implements the previous BASIC counter class in C++. Only fragments of the code are shown here, a listing of the entire sample can be found in the `/sqllib/samples/ole` directory.

The increment method is described in the Object Description Language as part of the counter interface description:

```
interface ICounter : IDispatch
{
    ...
    HRESULT increment([out] long    *out,
                      [out] short   *outnull,
                      [out] BSTR    *sqlstate,
                      [in]  BSTR    *fname,
                      [in]  BSTR    *fspecname,
                      [out] BSTR    *msgtext,
                      [in,out] SAFEARRAY (unsigned char) *spad,
                      [in]  long    *calltype);
    ...
}
```

The COM counter class definition in C++ includes the declaration of the `increment` method as well as `nbrOfInvoke`:

```
class FAR CCounter : public ICounter
{
        ...
        STDMETHODIMP CCounter::increment(long    *out,
                                         short   *outnull,
                                         BSTR    *sqlstate,
                                         BSTR    *fname,
                                         BSTR    *fspecname,
                                         BSTR    *msgtext,
                                         SAFEARRAY **spad,
                                         long *calltype );
        long nbrOfInvoke;
        ...
};
```

The C++ implementation of the method is similar to the BASIC code:

```
STDMETHODIMP CCounter::increment(long    *out,
                                 short   *outnull,
                                 BSTR    *sqlstate,
                                 BSTR    *fname,
                                 BSTR    *fspecname,
                                 BSTR    *msgtext,
                                 SAFEARRAY **spad,
                                 long *calltype)
{

  nbrOfInvoke = nbrOfInvoke + 1;
  *out = nbrOfInvoke;

  return NOERROR;
};
```

In the above example, sqlstate and msgtext are [out] parameters of type BSTR*, that is, DB2 passes a pointer to NULL to the UDF. To return values for these parameters, the UDF allocates a string and returns it to DB2 (for example, *sqlstate = SysAllocString (L"01H00")), and DB2 frees the memory. The parameters fname and fspecname are [in] parameters. DB2 allocates the memory and passes in values which are read by the UDF, and then DB2 frees the memory.

The class factory of the counter class creates counter objects. You can register the class factory as a single-use or multi-use object (not shown in this example).

```
STDMETHODIMP CCounterCF::CreateInstance(IUnknown FAR* punkOuter,
                                        REFIID riid,
                                        void FAR* FAR* ppv)
{

  CCounter *pObj;
        ...
  // create a new counter object
  pObj = new CCounter;
        ...
};
```

The counter class is implemented as a local server, and it is registered under the progId bert.ccounter. The following CREATE FUNCTION statement registers a UDF ccounter with the increment method as an external implementation:

```
CREATE FUNCTION ccounter () RETURNS integer
  EXTERNAL NAME 'bert.ccounter!increment'
  LANGUAGE OLE
  FENCED
  SCRATCHPAD
  FINAL CALL
  NOT DETERMINISTIC
  NULL CALL
  PARAMETER STYLE DB2SQL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

For the following query:

```
SELECT INT1, CCOUNTER() AS COUNT, INT1/CCOUNTER() AS DIV FROM TEST
```

The results are exactly the same as in the previous example:

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19

  5 record(s) selected.
```

While processing this query, DB2 creates two different instances of class `counter`, one for each UDF reference in the query. The two instances are reused for the entire query as the scratchpad option is specified in the `ccounter` UDF registration.

Example: Mail OLE Automation Table Function in BASIC

The following example implements a class using Microsoft Visual BASIC that exposes a public method `list` to retrieve message header information and the partial message text of messages in Microsoft Exchange. The method implementation employs OLE Messaging which provides an OLE automation interface to MAPI (Messaging API).

```
Description="Mail OLE Automation Table Function"
Module=MainModule; MainModule.bas
Class=Header; Header.cls
ExeName32="tfmapi.dll"
Name="TFMAIL"

VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Header"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit

Dim MySession As Object
Dim MyMsgColl As Object
Dim MyMsg As Object
Dim CurrentSender As Object
Dim name As Variant
Const SQL_TF_OPEN = -1
Const SQL_TF_CLOSE = 1
Const SQL_TF_FETCH = 0


Public Sub List(timereceived As Date, subject As String, size As Long, _
                text As String, ind1 As Integer, ind2 As Integer, _
                ind3 As Integer, ind4 As Integer, sqlstate As String, _
                fname As String, fspecname As String, msg As String, _
                scratchpad() As Byte, calltype As Long)


    If (calltype = SQL_TF_OPEN) Then

       Set MySession = CreateObject("MAPI.Session")
       MySession.Logon ProfileName:="Profile1"
       Set MyMsgColl = MySession.Inbox.Messages

       Set MyMsg = MyMsgColl.GetFirst

    ElseIf (calltype = SQL_TF_CLOSE) Then

       MySession.Logoff
       Set MySession = Nothing

    Else

      If (MyMsg Is Nothing) Then

         sqlstate = "02000"

      Else

         timereceived = MyMsg.timereceived
         subject = Left(MyMsg.subject, 15)
         size = MyMsg.size
         text = Left(MyMsg.text, 30)

         Set MyMsg = MyMsgColl.GetNext

      End If
    End If
End Sub
```

On the table function OPEN call, the CreateObject statement creates a mail session, and the logon method logs on to the mail system (user name and password issues are neglected). The message collection of the mail inbox is used to retrieve the first message. On the FETCH calls, the message header information and the first 30 characters of the current message are assigned to the table function output parameters. If no messages are left, SQLSTATE 02000 is returned. On the CLOSE call, the example logs off and sets the session object to nothing, which releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION MAIL()
    RETURNS TABLE (TIMERECIEVED DATE,
                   SUBJECT VARCHAR(15),
                   SIZE INTEGER,
                   TEXT VARCHAR(30))
    EXTERNAL NAME 'tfmail.header!list'
    LANGUAGE OLE
    PARAMETER STYLE DB2SQL
    NOT DETERMINISTIC
    FENCED
    NULL CALL
    SCRATCHPAD
    FINAL CALL
    NO SQL
    EXTERNAL ACTION
    DISALLOW PARALLEL;
```

Following is a sample query:

```
SELECT * FROM TABLE (MAIL()) AS M

TIMERECEIVED SUBJECT         SIZE        TEXT
------------ --------------- ----------- ------------------------------
01/18/1997   Welcome!               3277 Welcome to Windows Messaging!
01/18/1997   Invoice                1382 Please process this invoice. T
01/19/1997   Congratulations        1394 Congratulations to the purchas

  3 record(s) selected.
```

Debugging your UDF

It is important to debug your UDF in an environment where you cannot harm the database. You should do your testing on a test database instance until you are absolutely sure your UDF is correct. This is especially true if you plan to define your UDF as NOT FENCED, as a problematic not-fenced UDF can bring down an entire instance and this affects all the users of database. This risk is minimized by using a test instance for debugging. This is because DB2 may not fail gracefully in the case of certain actions by NOT FENCED UDFs, for example, an erroneous modification of storage.

DB2 does check for certain types of limited actions that erroneously modify storage (for example, if the UDF moves a few too many characters to a scratchpad or to the result buffer). In that case, DB2 returns an error, SQLCODE -443 (SQLSTATE 39501), if it detects such a malfunction. DB2 is also designed to fail gracefully in the event of an abnormal termination of a UDF with SQLCODE -430 (SQLSTATE 38503), or a user interrupt of the UDF's processing with SQLCODE -431 (SQLSTATE 38504).

There is no known scenario where a FENCED UDF can harm a database. That is why FENCED is the default, and why DB2 demands a special authority to define a UDF as NOT FENCED.

For security and database integrity reasons, it is important to protect the body of your UDF, once it is debugged and defined to DB2. This is particularly true if your UDF is defined as NOT FENCED. If either by accident or with malicious intent, anyone (including yourself) overwrites an operational UDF with code that is not debugged, the UDF could conceivably destroy the database if it is not-fenced, or compromise security.

Unfortunately, there is no easy way to run a source-level debugger on a UDF. There are several reasons for this:

- The timing makes it difficult to start the debugger at a time when the UDF is in storage and available
- The UDF runs in a database process with a special UID and the user is not permitted to attach to this process.

Note that valuable debugging tools such as `printf()` do not normally work as debugging aids for your UDF, because the UDF normally runs in a background process where `stdout` has no meaning. As an alternative to using `printf()`, it may be possible for you to instrument your UDF with file output logic, and for debugging purposes write indicative data and control information to a file.

Another technique to debug your UDF is to write a driver program for invoking the UDF outside the database environment. With this technique, you can invoke the UDF with all kinds of marginal or erroneous input arguments to attempt to provoke it into misbehaving. In this environment, it is not a problem to use `printf()` or a source level debugger.

Chapter 8. Using the Active DBMS Capabilities

In order to change your database manager from a passive system to an active one, use the capabilities embodied in a trigger function. A *trigger* defines a set of actions that are activated or *triggered* by, an update operation on a specified base table. These actions may cause other changes to the database, perform operations outside DB2 (for example, send an e-mail or write a record in a file), raise an exception to prevent the update operation from taking place, and so on. This chapter discusses the following topics which you need to understand to use triggers effectively in your applications:

- Why Use Triggers?
- Overview of a Trigger
- Trigger Event
- Set of Affected Rows
- Trigger Granularity
- Trigger Activation Time
- Transition Variables
- Transition Tables
- Triggered Action
- Trigger Cascading
- Interactions with Referential Constraints
- Ordering of Multiple Triggers
- Synergy Between Triggers, Constraints, UDTs, UDFs, and LOBs

Why Use Triggers?

You can use triggers to support general forms of integrity such as business rules. For example, your business may wish to refuse orders that exceed its customers' credit limit. A trigger can be used to enforce this constraint. In general, triggers are powerful mechanisms to capture *transitional* business rules. Transitional business rules are rules that involve different states of the data. For example, suppose a salary cannot be increased by more than 10 per cent. To check this rule, the value of the salary before and after the increase must be compared. For rules that do not involve more than one state of the data, check and referential integrity constraints may be more appropriate (see the *SQL Reference* for more information). Because of the declarative semantics of check and referential constraints, their use is recommended for constraints that are not transitional.

You can also use triggers for tasks such as automatically updating summary data. By keeping these actions as a part of the database and ensuring that they occur automatically, triggers enhance database integrity. For example, suppose you want to automatically track the number of employees managed by a company:

```
Tables: EMPLOYEE (as in Sample Tables)
  COMPANY_STATS (NBEMP, NBPRODUCT, REVENUE)
```

You can define two triggers:

- A trigger that increments the number of employees each time a new person is hired, that is, each time a new row is inserted into the table EMPLOYEE:

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

- A trigger that decrements the number of employees each time an employee leaves the company, that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

Specifically, you can use triggers to:

- Validate input data using the SIGNAL SQLSTATE SQL statement, the built-in RAISE_ERROR function, or invoke a UDF to return an SQLSTATE indicating that an error has occurred, if invalid data is discovered. Note that validation of non-transitional data is usually better handled by check and referential constraints. By contrast, triggers are appropriate for validation of transitional data, that is, validations which require comparisons between the value before and after an update operation.
- Automatically generate values for newly inserted rows (this is known as a *surrogate function*). That is, to implement user-defined default values, possibly based on other values in the row or values in other tables.
- Read from other tables for cross-referencing purposes.
- Write to other tables for audit-trail purposes.
- Support *alerts* (for example, through electronic mail messages).

Benefits and Value of Triggers

Using triggers in your database manager can result in:

1. **Faster application development**.

   Because triggers are stored in the relational database, the actions performed by triggers do not have to be coded in each application.

2. **Global enforcement of business rules**

   A trigger only has to be defined once, and then it can be used for any application that changes the table.

3. **Easier maintenance**

   If a business policy changes, only the corresponding trigger needs to change instead of each application program.

Overview of a Trigger

A trigger is a schema object associated with a base table and with an event that corresponds to an update operation on the base table (denoted as the *subject table*). The term *update operation* is used here to denote any change to the state of the subject table, regardless of whether that change was initiated by an SQL INSERT, UPDATE, DELETE, or an action of a referential constraint. When such an update operation (denoted as the *triggering operation*) is executed, the trigger is said to be activated. Once activated, the set of rows in the subject table that are affected by the update operation are determined and the action of the trigger is executed. During execution, the trigger action condition is tested and if satisfied, the triggered SQL statements are executed. Such an execution can take place before or after the update operation (or trigger event) and either only once for the update operation or for each row affected by the operation. A trigger can be defined using the CREATE TRIGGER statement. For details of the CREATE TRIGGER statement, see the *SQL Reference*.

When a trigger is created, the following items are defined for the trigger:

- The name.
- The name of the table that is the subject table.
- The trigger activation time (BEFORE or AFTER).
- The trigger event (INSERT, DELETE, or UPDATE).
- The old values transition variable, if any.
- The new values transition variable, if any.
- The old values transition table, if any.
- The new values transition table, if any.
- The granularity (FOR EACH STATEMENT or FOR EACH ROW).
- The triggered action of the trigger (including a triggered action condition and triggered SQL statement(s)).
- If the trigger event is UPDATE, then the trigger column list for the trigger event of the trigger, as well as an indication of whether the trigger column list was explicit or implicit.
- The trigger creation timestamp.
- The current function path.

Trigger Event

Every trigger is associated with an event. Triggers are activated when their corresponding event occurs in the database. This event occurs when the specified triggering SQL operation, either an UPDATE, INSERT, or DELETE (including those caused by actions of referential constraints), is performed on the subject table. For example:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

The above statement defines that the trigger `new_hire` is activated when an insert operation is performed on table `employee`.

Every event, and consequently, every trigger, can be associated with exactly one subject table and exactly one *triggering operation*. The triggering operations are:

Insert operation An insert operation can only be caused by an INSERT statement. Therefore, triggers are not activated when data is loaded using utilities that do not use INSERT, such as the LOAD command.

Update operation An update operation can be caused by an UPDATE statement or as a result of a referential constraint rule of ON DELETE SET NULL.

Delete operation A delete operation can be caused by a DELETE statement or as a result of a referential constraint rule of ON DELETE CASCADE.

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 only activated if the update operation attempts to update any of the specified columns. This provides a further refinement of the event that activates the trigger. For example, the following trigger, REORDER, will be activated only if an update operation is performed on the columns ON_HAND or MAX_STOCKED, of the table PARTS.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
  VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
  END
```

Set of Affected Rows

A triggering SQL operation defines a set of rows in the subject table that is affected by that SQL operation. For example, suppose you run the following SQL UPDATE statement on the parts table:

```
UPDATE PARTS
  SET ON_HAND = ON_HAND + 100
  WHERE  PART_NO > 15000
```

The set of affected rows for an update trigger contains all the rows in the parts table whose *part_no* is greater than 15 000.

Trigger Granularity

When a trigger is activated, it runs according to its granularity as follows:

FOR EACH ROW It runs as many times as the number of rows in the set of affected rows.

FOR EACH STATEMENT It runs once for the triggering operation.

If the set of affected rows is empty (that is, in the case of a searched UPDATE or DELETE in which the WHERE clause did not qualify any rows), a FOR EACH ROW trigger does not run. But a FOR EACH STATEMENT trigger still runs once.

For example, keeping a count of number of employees can be done using FOR EACH ROW.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

You can achieve the same affect with one update by using a granularity of FOR EACH STATEMENT.

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  REFERENCING NEW_TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  UPDATE COMPANY_STATS
  SET NBEMP = NBEMP + (SELECT COUNT(*) FROM NEWEMPS)
```

Note: A granularity of FOR EACH STATEMENT is not supported for BEFORE triggers (discussed in “Trigger Activation Time”).

Trigger Activation Time

The *trigger activation time* specifies when the trigger should be activated. That is, either BEFORE or AFTER the activation of the triggering SQL operation of its event. For example, the activation time of the following trigger is AFTER the INSERT operation on `employee`.

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

If the activation time is BEFORE, the triggered actions are activated for each row in the set of affected rows before the triggering SQL operation is performed. Note that BEFORE triggers must have a granularity of FOR EACH ROW.

If the activation time is AFTER, the triggered actions are activated for each row in the set of affected rows or for the statement, depending on the trigger granularity. This occurs after the triggering SQL operation is performed, and after the checking of all constraints that may be affected by the triggering SQL operation, including actions of referential constraints. Note that AFTER triggers can have a granularity of either FOR EACH ROW or FOR EACH STATEMENT.

The different activation times of triggers reflect different purposes of triggers. Basically, BEFORE triggers are an extension to the constraint subsystem of the database management system. Therefore, you generally use them to:

- Perform validation of input data,

- Automatically generate values for newly inserted rows
- Read from other tables for cross-referencing purposes.

BEFORE triggers are not used for further modifying the database because they are activated before the triggering SQL operation is applied to the database. Consequently, they are activated before integrity constraints are checked and may be violated by the triggering SQL operation.

Conversely, you can view AFTER triggers as a module of application logic that runs in the database every time a specific event occurs. As a part of an application, AFTER triggers always see the database in a consistent state. Note that they are run after the integrity constraints that may be violated by the triggering SQL operation have been checked. Consequently, you can use them mostly to perform operations that an application can also perform. For example:

- Perform follow on update operations in the database
- Perform actions outside the database, for example, to support alerts. Note that actions performed outside the database are not rolled back if the trigger is rolled back.

Because of the different nature of BEFORE and AFTER triggers, a different set of SQL operations can be used to define the triggered actions of BEFORE and AFTER triggers. For example, update operations are not allowed in BEFORE triggers because there is no guarantee that integrity constraints will not be violated by the triggering SQL operation. The set of SQL operations you can specify in BEFORE and AFTER triggers are described in “Triggered Action” on page 357. Similarly, different trigger granularities are supported in BEFORE and AFTER triggers. For example, FOR EACH STATEMENT is not allowed in BEFORE triggers because there is no guarantee that constraints will not be violated by the triggering SQL operation, which would, in turn, result in failure of the operation.

Transition Variables

When you carry out a FOR EACH ROW trigger, it may be necessary to refer to the value of columns of the row in the set of affected rows, for which the trigger is currently executing. Note that to refer to columns in tables in the database (including the subject table), you can use regular SELECT statements. A FOR EACH ROW trigger may refer to the columns of the row for which it is currently executing by using two transition variables that you can specify in the REFERENCING clause of a CREATE TRIGGER statement. There are two kinds of transition variables, which are specified as *OLD* and *NEW*, together with a correlation-name. They have the following semantics:

OLD correlation-name Specifies a correlation name which captures the original state of the row, that is, before the triggering SQL operation is applied to the database.

NEW correlation-name Specifies a correlation name which captures the value that is, or was, used to update the row in the database when the triggering SQL operation is applied to the database.

Consider the following example:

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED
  AND N_ROW.ORDER_PENDING = 'N')
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
    UPDATE PARTS SET PARTS.ORDER_PENDING = 'Y'
    WHERE PARTS.PARTNO = N_ROW.PARTNO;
  END
```

Based on the definition of the OLD and NEW transition variables given above, it is clear that not every transition variable can be defined for every trigger. Transition variables can be defined depending on the kind of trigger event:

UPDATE An UPDATE trigger can refer to both OLD and NEW transition variables.

INSERT An INSERT trigger can only refer to a NEW transition variable because before the activation of the INSERT operation, the affected row does not exist in the database. That is, there is no original state of the row that would define old values before the triggering SQL operation is applied to the database.

DELETE A DELETE trigger can only refer to an OLD transition variable because there are no new values specified in the delete operation.

Note: Transition variables can only be specified for FOR EACH ROW triggers. In a FOR EACH STATEMENT trigger, a reference to a transition variable is not sufficient to specify to which of the several rows in the set of affected rows the transition variable is referring.

Transition Tables

In both FOR EACH ROW and FOR EACH STATEMENT triggers, it may be necessary to refer to the whole set of affected rows. This is necessary, for example, if the trigger body needs to apply aggregations over the set of affected rows (for example, MAX, MIN, or AVG of some column values). A trigger may refer to the set of affected rows by using two transition tables that can be specified in the REFERENCING clause of a CREATE TRIGGER statement. Just like the transition variables, there are two kinds of transition tables, which are specified as OLD_TABLE and NEW_TABLE together with a *table-name*, with the following semantics:

OLD_TABLE table-name Specifies the name of the table which captures the original state of the set of affected rows (that is, before the triggering SQL operation is applied to the database).

NEW_TABLE table-name Specifies the name of the table which captures the value that is used to update the rows in the database when the triggering SQL operation is applied to the database.

For example:

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW_TABLE AS N_TABLE
  NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN ((SELECT AVG (ON_HAND) FROM N_TABLE) > 35)
  BEGIN ATOMIC
    VALUES(INFORM_SUPERVISOR(N_ROW.PARTNO,
                             N_ROW.MAX_STOCKED,
                             N_ROW.ON_HAND));
  END
```

Note that NEW_TABLE always has the full set of updated rows, even on a FOR EACH ROW trigger. When a trigger acts on the table on which the trigger is defined, NEW_TABLE contains the changed rows from the statement that activated the trigger. However, NEW_TABLE does not contain the changed rows that were caused by statements within the trigger, as that would cause a separate activation of the trigger.

The transition tables are read-only. The same rules that define the kinds of transition variables that can be defined for which trigger event, apply for transition tables:

UPDATE An UPDATE trigger can refer to both OLD_TABLE and NEW_TABLE transition tables.

INSERT An INSERT trigger can only refer to a NEW_TABLE transition table because before the activation of the INSERT operation the affected rows do not exist in the database. That is, there is no *original state of the rows* that defines old values before the triggering SQL operation is applied to the database.

DELETE A DELETE trigger can only refer to an OLD transition table because there are no new values specified in the delete operation.

Note: It is important to observe that transition tables can be specified for both granularities of AFTER triggers: FOR EACH ROW and FOR EACH STATEMENT.

The scope of the OLD_TABLE and NEW_TABLE *table-name* is the trigger body. In this scope, this name takes precedence over the name of any other table with the same unqualified *table-name* that may exist in the schema. Therefore, if the OLD_TABLE or NEW_TABLE *table-name* is for example, X, a reference to X (that is, an unqualified X) in the FROM clause of a SELECT statement will always refer to the transition table even if there is a table named X in the in the schema of the trigger creator. In this case, the user has to make use of the fully qualified name in order to refer to the table X in the schema.

Triggered Action

The activation of a trigger results in the running of its associated triggered action. Every trigger has exactly one triggered action which, in turn, has two components:

- An optional *triggered action condition* or WHEN clause
- A set of *triggered SQL statement(s)*.

The triggered action condition defines whether or not the set of triggered statements are performed for the row or for the statement for which the triggered action is executing. The set of triggered statements define the set of actions performed by the trigger in the database as a consequence of its event having occurred.

For example, the following trigger action specifies that the set of triggered SQL statements should only be activated for rows in which the value of the on_hand column is less than ten per cent of the value of the max_stocked column. In this case, the set of triggered SQL statements is the invocation of the issue_ship_request function.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL

  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
  END
```

Triggered Action Condition

As explained in “Triggered Action,” the *triggered action condition* is an optional clause of the triggered action which specifies a search condition that must evaluate to *true* to run SQL statements within the triggered action. If the WHEN clause is omitted, then the SQL statements within the triggered action are always executed.

The triggered action condition is evaluated once for each row if the trigger is a FOR EACH ROW trigger, and once for the statement if the trigger is a FOR EACH STATEMENT trigger.

This clause provides further control that you can use to fine tune the actions activated on behalf of a trigger. An example of the usefulness of the WHEN clause is to enforce a data dependent rule in which a triggered SQL statement is activated only if the incoming value falls inside or outside of a certain range.

Triggered SQL Statements

The set of triggered SQL statements carries out the *real* actions caused by activating a trigger. As described previously, not every SQL operation is meaningful in every trigger. Depending on whether the trigger activation time is BEFORE or AFTER, different kinds of operations may be appropriate as a triggered SQL statement.

For a list of triggered SQL statements, and additional information on BEFORE and AFTER triggers, see the *SQL Reference*.

In most cases, if any triggered SQL statement returns a negative return code, the triggering SQL statement together with all trigger and referential constraint actions are rolled back, and an error is returned: SQLCODE -723 (SQLSTATE 09000). The trigger name, SQLCODE, SQLSTATE and many of the tokens from the failing triggered SQL statement are returned. Error conditions occurring when triggers are running that are critical or roll back the entire unit of work are not returned using SQLCODE -723 (SQLSTATE 09000).

Functions Within SQL Triggered Statement

Functions, including user-defined functions (UDFs), may be invoked within a triggered SQL statement. Consider the following example:,

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES (ISSUE_SHIP_REQUEST (N_ROW.MAX_STOCKED - N_ROW.ON_HAND,
                                N_ROW.PARTNO));
  END
```

When a triggered SQL statement contains a function invocation with an unqualified function name, the function invocation is resolved based on the function path at the time of creation of the trigger. For details on the resolution of functions, see the *SQL Reference*.

UDFs are written in either the C or C++ programming language. This enables control of logic flows, error handling and recovery, and access to system and library functions. (See Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285 for a description of UDFs.) This capability allows a triggered action to perform non-SQL types of operations when a trigger is activated. For example, such a UDF could send an electronic mail message and thereby act as an alert mechanism. External actions, such as messages, are not under commit control and will be run regardless of success or failure of the rest of the triggered actions.

Also, the function can return an SQLSTATE that indicates an error has occurred which results in the failure of the triggering SQL statement. This is one method of implementing user-defined constraints. (Using a SIGNAL SQLSTATE statement is the other.) In order to use a trigger as a means to check complex user-defined constraints, you can use the RAISE_ERROR built-in function in a triggered SQL statement. This function can be used to return a user-defined SQLSTATE (SQLCODE -438) to applications. For details on invocation and use of this function, see the *SQL Reference*.

For example, consider some rules related to the HIREDATE column of the EMPLOYEE table, where HIREDATE is the date that the employee starts working.

- HIREDATE must be date of insert or a future date
- HIREDATE cannot be more than 1 year from date of insert.
- If HIREDATE is between 6 and 12 months from date of insert, notify personnel manager using a UDF called send_note.

The following trigger handles all of these rules on INSERT:

```
CREATE TRIGGER CHECK_HIREDATE
  NO CASCADE BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
  VALUES CASE
    WHEN NEW_EMP.HIREDATE < CURRENT DATE
      THEN RAISE_ERROR('85001', 'HIREDATE has passed')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 10000.
      THEN RAISE_ERROR('85002', 'HIREDATE too far out')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 600.
      THEN SEND_MOTE('persmgr',NEW_EMP.EMPNO,'late.txt')
    END;
  END
```

Trigger Cascading

When you run a triggered SQL statement, it may cause the event of another, or even the same, trigger to occur, which in turn, causes the other, (or a second instance of the same) trigger to be activated. Therefore, activating a trigger can cascade the activation of one or more other triggers.

The run-time depth level of trigger cascading supported is 16. If a trigger at level 17 is activated, SQLCODE -724 (SQLSTATE 54038) will be returned and the triggering statement will be rolled back.

Interactions with Referential Constraints

As described above, the triggering SQL operation can be the result of changes due to referential constraint enforcement. For example, given two tables DEPT and EMP, if deleting or updating DEPT causes propagated deletes or updates to EMP by means of referential integrity constraints, then delete or update triggers defined on EMP become activated as a result of the referential constraint defined on DEPT. The triggers on EMP are run either BEFORE or AFTER the deletion (in the case of ON DELETE CASCADE) or update of rows in EMP (in the case of ON DELETE SET NULL), depending on their activation time.

Ordering of Multiple Triggers

When triggers are defined using the CREATE TRIGGER statement, their creation time is registered in the database in form of a timestamp. The value of this timestamp is subsequently used to order the activation of triggers when there is more than one trigger that should be run at the same time. For example, the timestamp is used when there is more than one trigger on the same subject table with the same event and the same activation time. The timestamp is also used when there is one or more AFTER triggers that are activated by the triggering SQL operation and referential constraint actions caused directly or indirectly (that is, recursively by other referential constraints) by the triggering SQL operation. Consider the following two triggers:

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

CREATE TRIGGER NEW_HIRED_DEPT
   AFTER INSERT ON EMPLOYEE
   REFERENCING NEW AS EMP
   FOR EACH ROW MODE DB2SQL
     BEGIN ATOMIC
       UPDATE DEPTS
       SET NBEMP = NBEMP + 1
       WHERE DEPT_ID = EMP.DEPT_ID;
     END;
```

The above triggers are activated when you run an INSERT operation on the employee table. In this case, the timestamp of their creation defines which of the above two triggers is activated first.

The activation of the triggers is conducted in ascending order of the timestamp value. Thus, a trigger that is newly added to a database runs after all the other triggers that are previously defined.

Old triggers are activated before new triggers to ensure that new triggers can be used as *incremental* additions to the changes that affect the database. For example, if a triggered SQL statement of trigger T1 inserts a new row into a table T, a triggered SQL statement of trigger T2 that is run after T1 can be used to update the same row in T with specific values. By activating triggers in ascending order of creation, you can ensure that the actions of new triggers run on a database that reflects the result of the activation of all old triggers.

Synergy Between Triggers, Constraints, UDTs, UDFs, and LOBs

“Synergy Between UDTs, UDFs, and LOBs” on page 280 describes how to combine UDTs, UDFs, and LOBs to represent and manipulate large, complex structures. The following section describes how to exploit triggers and constraints to complete the modeling of such application structures. With triggers, you can:

- Extract information from these structures to keep them explicitly in columns of tables (instead of hidden within the structure)
- Define the integrity rules that govern these structures in the application domain

- Express important actions that need to be taken under certain values of the structures.

Extracting Information

In “Synergy Between UDTs, UDFs, and LOBs” on page 280, the complete electronic mail is stored within the column message of the ELECTRONIC_MAIL table. To manipulate the electronic mail, UDFs were used to extract information from the message column every time such information was required within an SQL statement.

Notice that the queries do not extract information once and store it explicitly as columns of tables. If this was done, it would increase the performance of the queries, not only because the UDFs are not invoked repeatedly, but also because you can then define indexes on the extracted information.

Using triggers, you can extract this information whenever new electronic mail is stored in the database. To achieve this, add new columns to the ELECTRONIC_MAIL table and define a BEFORE trigger to extract the corresponding information as follows:

```
ALTER TABLE ELECTRONIC_MAIL
  ADD COLUMN SENDER    VARCHAR (200)
  ADD COLUMN RECEIVER  VARCHAR (200)
  ADD COLUMN SENT_ON   DATE
  ADD COLUMN SUBJECT   VARCHAR (200)

CREATE TRIGGER EXTRACT_INFO
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.SENDER = SENDER(N.MESSAGE);
    SET N.RECEIVER = RECEIVER(N.MESSAGE);
    SET N.SENT_ON = SENDING_DATE(N.MESSAGE);
    SET N.SUBJECT = SUBJECT(N.MESSAGE);
  END
```

Now, whenever new electronic mail is inserted into the message column, its sender, its receiver, the date on which it was sent, and its subject are extracted from the message and stored in separate columns.

Preventing Operations on Tables

Suppose you want to prevent mail you sent, which was undelivered and returned to you (perhaps because the e-mail address was incorrect), from being stored in the e-mail’s table.

To do so, you need to prevent the execution of certain SQL INSERT statements. There are two ways to do this:

- Define a BEFORE trigger that raises an error whenever the subject of an e-mail is *undelivered mail*:

```
CREATE TRIGGER BLOCK_INSERT
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (SUBJECT(N.MESSAGE) = 'undelivered mail')
  BEGIN ATOMIC
    SIGNAL SQLSTATE '85101' ('Attempt to insert undelivered mail');
  END
```

- Define a check constraint forcing values of the new column subject to be different from *undelivered mail*:

  ```
  ALTER TABLE ELECTRONIC_MAIL
    ADD CONSTRAINT NO_UNDELIVERED
    CHECK (SUBJECT <> 'undelivered mail')
  ```

Because of the advantages of the declarative nature of constraints, the constraint should generally be defined instead of the trigger.

Defining Business Rules

Suppose your company has the policy that all e-mail dealing with customer complaints must have Mr. Nelson, the marketing manager, in the carbon copy (CC) list. Because this is a rule, you might want to express it as a constraint such as one of the following (assuming the existence of a CC_LIST UDF to check it):

```
ALTER TABLE ELECTRONIC_MAIL ADD
  CHECK (SUBJECT <> 'Customer complaint' OR
         CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 1)
```

However, such a constraint prevents the insertion of e-mail dealing with customer complaints that do not have the marketing manager in the cc list. This is certainly not the intent of your company’s business rule. The intent is to forward to the marketing manager any e-mail dealing with customer complaints that were not copied to the marketing manager. Such a business rule can only be expressed with a trigger because it requires taking actions that cannot be expressed with declarative constraints. The trigger assumes the existence of a SEND_NOTE function with parameters of type E_MAIL and character string.

```
CREATE TRIGGER INFORM_MANAGER
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (N.SUBJECT = 'Customer complaint' AND
    CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 0)
  BEGIN ATOMIC
    VALUES(SEND_NOTE(N.MESSAGE, 'nelson@vnet.ibm.com'));
  END
```

Defining Actions

Now assume that your general manager wants to keep the names of customers who have sent three or more complaints in the last 72 hours in a separate table . The general manager also wants to be informed whenever a customer name is inserted in this table more than once.

To define such actions, you define:

- An UNHAPPY_CUSTOMERS table:

```
CREATE TABLE UNHAPPY_CUSTOMERS (
  NAME           VARCHAR (30),
  EMAIL_ADDRESS  VARCHAR (200),
  INSERTION_DATE DATE)
```

- A trigger to automatically insert a row in UNHAPPY_CUSTOMERS if 3 or more messages were received in the last 3 days (assumes the existence of a CUSTOMERS table that includes a NAME column and an E_MAIL_ADDRESS column):

```
CREATE TRIGGER STORE_UNHAPPY_CUST
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (3 <= (SELECT COUNT(*)
              FROM ELECTRONIC_MAIL
              WHERE SENDER = N.SENDER
                AND SENDING_DATE(MESSAGE) > CURRENT DATE - 3 DAYS)
       )
  BEGIN ATOMIC
    INSERT INTO UNHAPPY_CUSTOMERS
    VALUES ((SELECT NAME
    FROM CUSTOMERS
    WHERE E_MAIL_ADDRESS = N.SENDER), N.SENDER, CURRENT DATE);
  END
```

- A trigger to send a note to the general manager if the same customer is inserted in UNHAPPY_CUSTOMERS more than once (assumes the existence of a SEND_NOTE function that takes 2 character strings as input):

```
CREATE TRIGGER INFORM_GEN_MGR
  AFTER INSERT ON UNHAPPY_CUSTOMERS
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (1 <(SELECT COUNT(*)
            FROM UNHAPPY_CUSTOMERS
            WHERE EMAIL_ADDRESS = N.EMAIL_ADDRESS)
       )
  BEGIN ATOMIC
    VALUES(SEND_NOTE('Check customer:' CONCAT N.NAME,
                     'bigboss@vnet.ibm.com'));
  END
```

Chapter 9. Programming in Complex Environments

This chapter discusses advanced topics that you need to address if your application runs in an environment with any of the following complications:

- National Language Support Considerations
- Japanese and Traditional-Chinese EUC Code Set Considerations
- Considerations for Distributed Unit of Work (DUOW)
- Accessing DRDA Servers
- Multiple Thread Database Access
- Concurrent Transactions
- X/Open XA Interface Programming Considerations
- Working with Large Volumes of Data Across a Network

National Language Support Considerations

This section describes National Language Support (NLS) support issues that you must consider for your applications. The major topics discussed are:

- Collating Sequences
- Deriving Code Page Values
- Deriving Locales in Application Programs
- Programming Considerations
- Conversion Between Different Code Pages

Collating Sequences

The database manager compares character data using a *collating sequence*. This is an ordering for a set of characters that determines whether a particular character sorts higher, lower, or the same as another.

Note: Character string data defined with the FOR BIT DATA attribute, or BLOB data, is sorted using the binary sort sequence.

For example, a collating sequence can be used to indicate that lowercase and uppercase versions of a particular character are to be sorted equally.

The database manager allows databases to be created with custom collating sequences. The following sections help you determine and implement a particular collating sequence for a database.

Overview

In a database, each single-byte character is represented internally as a unique number between 0 and 255, (in hexadecimal notation, between `X'00'` and `X'FF'`). This number is referred to as the *code point* of the character. A collating sequence is a mapping between the code point and the desired position of each character in a sorted sequence. The numeric value of the position is called the *weight* of the character in the collating sequence. The simplest collating sequence is one where the weights are identical to the code points. This is called the *identity sequence*.

For example, consider the characters B (X'42'), and b (X'62'). If, according to the collating sequence table, they both have a sort weight of X'42' (B), then they collate the same. If the sort weight for B is X'9E' and the sort weight for b is X'9D', then b will be sorted before B. Actual weights depend on the collating sequence table used which depends on the code set and locale. Note that a collating sequence table is not the same as a code page table which defines code points.

Consider the following example. In ASCII, the characters A through Z are represented by X'41' through X'5A'. To describe a collating sequence where these are sorted in order, and consecutively (no intervening characters), you can write X'41', X'42', ...X'59', X'5A'.

For multi-byte characters, the hexadecimal value of the multi-byte character is also used as the weight. For example, X'8260', X'8261' are the code points for double byte character A and B. In this case, you can write X'8260', X'8261' as the collating sequence for double byte characters A and B. These are also the code points for A and B.

The values of the weights in a collating sequence need not be unique. For example, you could give uppercase letters and their lowercase equivalents the same weight.

Specifying the collating sequence can be simplified if a collating sequence provides weights for all 256 code points. The weight of each character can be determined using the code point of the character. This is the method used to specify a collating sequence for the database manager: a string of 256 bytes, where the *nth* byte (starting with 0) contains the weight of code point *n*.

In the case of multi-byte character sets, DB2 simply uses the code point as the collating value. Multi-byte characters therefore sort the way they appear in their code point table.

Character Comparisons: Once a collating sequence is established, character comparison is performed by comparing the weights of two characters, instead of directly comparing their code point values.

If weights that are not unique are used, characters that are not identical may compare equally. Because of this, string comparison must be a two-phase process:

1. Compare the characters of each string based on their weights.
2. If step 1 yielded equality, compare the characters of each string based on their code point values.

If the collating sequence contains 256 unique weights, only the first step is performed. If the collating sequence is the identity sequence only the second step is performed. In either case, there is a performance benefit.

For more information on character comparisons, see the *SQL Reference*.

Case Independent Comparisons: To perform character comparisons that are independent of whether they are upper or lower case, you can use the TRANSLATE function to select and compare mixed case column data by translating it to upper case, but only for the purposes of comparison. Consider the following data:

```
Abel
abels
ABEL
abel
ab
Ab
```

For the following select statement:

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

you would receive the following results:

```
ab
Ab
abel
Abel
ABEL
abels
```

Note: You could also set the select as in the following view v1, and then make all your comparisons against the view (in upper case) and your inserts into the table in mixed case:

```
CREATE VIEW v1 AS SELECT TRANSLATE(c1) FROM t1
```

At the database level, you can set the *collating sequence* as part of the CREATE DATABASE API . This allows you to decide if 'a' is processed before 'A', or if 'A' is processed after 'a', or if they are processed with equal weighting. This will make them equal when collating or sorting using the ORDER BY clause. If you have two values of 'a' and 'A', 'A' will always come before 'a', because in all senses they are equal, so the only difference upon which to sort is the hexadecimal value.

Thus if you issue SELECT c1 FROM t1 WHERE c1 LIKE 'ab%', you receive the following output:

```
ab
abel
abels
```

If you issue SELECT c1 FROM t1 WHERE c1 LIKE 'A%', you receive the following output:

```
Abel
Ab
ABEL
```

If you issue SELECT c1 FROM t1 ORDER BY c1, you receive the following:

```
ab
Ab
abel
Abel
ABEL
abels
```

Thus, you may want to consider using the scalar function TRANSLATE(), as well as the CREATE DATABASE API. Note that you can only specify a collating sequence using the CREATE DATABASE API. You cannot specify a collating sequence from the Command Line Processor. For information on the TRANSLATE() function, see the *SQL Reference*. For information on the CREATE DATABASE API see the *API Reference*.

You can also use the UCASE function as follows, but note that DB2 performs a table scan instead of using an index for the select:

```
    SELECT * FROM EMP WHERE UCASE(JOB) = 'NURSE'
```

Specifying a Collating Sequence

The collating sequence for a database is specified at database creation time. Once the database has been created, the collating sequence cannot be changed.

The CREATE DATABASE API accepts a data structure called the Database Descriptor Block (SQLEDBDESC). You can define your own collating sequence within this structure.

To specify a collating sequence for a database:

- Pass the desired SQLEDBDESC structure, or
- Pass a NULL pointer. The collating sequence of the operating system (based on current country code and code page) is used. This is the same as specifying SQLDBCSS equal to SQL_CS_SYSTEM (0).

The SQLEDBDESC structure contains:

SQLDBCSS A 4-byte integer indicating the source of the database collating sequence. Valid values are:

SQL_CS_SYSTEM The collating sequence of the operating system (based on current country code and code page) is used.

SQL_CS_USER The collating sequence is specified by the value in the SQLDBUDC field.

SQL_CS_NONE The collating sequence is the identity sequence. Strings are compared byte for byte, starting with the first byte, using a simple binary comparison.

Note: These constants are defined in the SQLENV include file.

SQLDBUDC A 256-byte field. The nth byte contains the sort weight of the nth character in the code page of the database. If SQLDBCSS is not equal to SQL_CS_USER, this field is ignored.

Sample Collating Sequences: Several sample collating sequences are provided (as include files) to facilitate database creation using the EBCDIC collating sequences instead of the default workstation collating sequence.

The collating sequences in these include files can be specified in the SQLDBUDC field of the SQLEDBDESC structure. They can also be used as models for the construction of other collating sequences.

For information on the include files that contain collating sequences, see the following sections:

- For C/C++, "Include Files" on page 421
- For COBOL, "Include Files" on page 453
- For FORTRAN, "Include Files" on page 477.

Other Concerns: Once a collating sequence is defined, all future character comparisons for that database will be performed with that collating sequence. Except for character data defined as FOR BIT DATA or BLOB data, the collating sequence will be used for all SQL comparisons and ORDER BY clauses, and also in setting up indexes and statistics. For more information on how the database collating sequence is used, see the section on *String Comparisons* in the *SQL Reference*, S10J-8165-00.

Potential problems may occur in the following cases:

- An application merges sorted data from a database with application data that was sorted using a different collating sequence.
- An application merges sorted data from one database with sorted data from another, but the databases have different collating sequences.
- An application makes assumptions about sorted data that are not true for the relevant collating sequence. For example, numbers collating lower than alphabetics might or might not be true for a particular collating sequence.

A final point to remember is that the results of any sort based on a direct comparison of characters will only match the results of a query ordered using an identity collating sequence.

Deriving Code Page Values

The **application code page** is derived from the active environment when the database connection is made. If the DB2CODEPAGE environment variable is set, its value is taken as the application code page. The **database code page** is derived from the value specified (explicitly or by default) at the time the database is created. The following defines how the *active environment* is determined in different operating environments, for example:

UNIX
In UNIX-based environments, the active environment is determined from the locale environment variables, which include information about language, territory and code set.

OS/2
In OS/2, primary and secondary code pages are specified in the `CONFIG.SYS` file. You can use the `chcp` command to display and dynamically change code pages within a given session.

DOS
In DOS, the active code page is determined by the value specified in the `COUNTRY` command in the `CONFIG.SYS` file. You can use the `chcp` command to display and dynamically change code pages within a given session.

Macintosh
For the Macintosh operating system, if the DB2CODEPAGE environment variable is not set, the Macintosh code page is derived from the Regional version code from the installed script.

Windows
For Windows, if the DB2CODEPAGE environment variable is not set, the Windows code page is derived from the country ID, as specified in the `iCountry` value in the `[intl]` section of the Windows WIN.INI file.

Windows 95
For Windows 95, if the DB2CODEPAGE environment variable is not set, the Windows 95 code page is derived from the ANSI code page setting in the Registry.

Windows NT
For Windows NT, if the DB2CODEPAGE environment variable is not set, the Windows NT code page is derived from the ANSI code page setting in the Registry.

For a complete list of environment mappings for code page values, see the Table 30 on page 553.

Deriving Locales in Application Programs

Locales are specific to UNIX-based operating systems. There are two locales:

- The environment locale allows you to specify the language, currency symbol, and so on, that you want to use.
- The program locale contains the current language, currency symbol, and so on, of a program that is executing.

When your program is started, it gets a default C locale. It does **not** get a copy of the environment locale. Your program has a few choices:

- Ignore the environment locale. Your program could hard code some options. For example, your program could set the language to *spanish* with the `setlocale()` function.
- Copy the environment locale to the program locale.

- Ignore the environment locale. Use whatever defaults you get from the operating system.

How DB2 Derives Locales

With UNIX, the active locale used by DB2 is determined from the LC_CTYPE portion of the locale. For details, see the NLS documentation for your operating system.

- If LC_CTYPE of the program locale has a value other than that of 'C', DB2 will use this value to determine the application code page by mapping it to its corresponding code page.
- If LC_CTYPE has the value of 'C' (the 'C' locale), DB2 will set the program locale according to the environment locale using the `setlocale()` function.
- If LC_CTYPE still has a value of 'C', DB2 will use its default locale for that platform. See either the *Building Applications for Windows and OS/2 Environments* or the *Building Applications for UNIX Environments* book for information on the default locale for that platform.
- If LC_CTYPE's value is no longer 'C', its new value will be used to map to a corresponding code page.

Programming Considerations

It is strongly recommended that applications be precompiled, bound, compiled, and executed using the same code page. This is because data conversions by the server can occur in both the bind and the execution phases. Users should ensure that the same conversion tables are used by binding and executing with the same active code page. For a discussion of how applications determine the active code page, see "Deriving Code Page Values" on page 369.

Any external data obtained by the application will be assumed to be in the application code page. This includes data obtained from a file or from user input. Make sure that data from sources outside the application uses the same code page as the application.

If you use host variables that use graphic data in your C or C++ applications, there are special precompiler, application performance, and application design issues you need to consider. For a detailed discussion of these considerations, see "Handling Graphic Host Variables" on page 440. If you deal with EUC code sets in your applications, refer to "Japanese and Traditional-Chinese EUC Code Set Considerations" on page 377 for guidelines that you should consider.

Coding SQL Statements

The coding of SQL statements is not language dependent. The SQL keywords must be typed as shown in this book, although they may be typed in uppercase, lowercase, or mixed case. The names of database objects, host variables and program labels that occur in an SQL statement cannot contain characters outside the extended character set supported by your code page. See the *SQL Reference* for more information about extended character sets.

Constant character strings in static SQL statements are converted at bind time, from the application code page to the database code page, and will be used at execution time in this database code page representation. To avoid such conversions if they are not desired, you can use host variables in place of string constants.

The server does not convert file names. To code a file name, either use the ASCII invariant set, or provide the path in the hexadecimal values that are physically stored in the file system.

In a multi-byte environment, there are four characters which are considered special that do not belong to the invariant character set. These characters are:

- The double-byte percentage and double-byte underscore characters used in LIKE processing. For further details concerning LIKE, refer to the *SQL Reference*.
- The double-byte space character, used for, among other things, blank padding in graphic strings.
- The double-byte substitution character, used as a replacement during character conversion when no mapping exists between a source code page and a target code page.

The code points for each of these characters, by code page is as follows:

Table 13. Code Points for Special Double-byte Characters

Code Page	Double-Byte Percentage	Double-Byte Underscore	Double-byte Space	Double-Byte Substitution Character
932	X'8193'	X'8151'	X'8140'	X'FCFC'
938	X'8193'	X'8151'	X'8140'	X'FCFC'
942	X'8193'	X'8151'	X'8140'	X'FCFC'
943	X'8193'	X'8151'	X'8140'	X'FCFC'
948	X'8193'	X'8151'	X'8140'	X'FCFC'
949	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
950	X'A248'	X'A1C4'	X'A140'	X'C8FE'
954	X'A1F3'	X'A1B2'	X'A1A1'	X'F4FE'
964	X'A2E8'	X'A2A5'	X'A1A1'	X'FDFE'
970	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
1381	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
1383	X'A3A5'	X'A3DF'	X'A1A1'	X'A1A1'
13488	X'FF05'	X'FF3F'	X'3000'	X'FFFD'

EUC Considerations: The DBCS substitution character is used to replace any EUC non-SBCS character as required. There is no concept of a three or four byte substitution character.

Coding Remote Stored Procedures and UDFs

When coding stored procedures that will be running remotely, the following considerations apply:

- Data in a stored procedure must be in the database code page.
- Data passed to or from a stored procedure using an SQLDA with a character data type must really contain character data. Numeric data and data structures must never be passed with a character type if the client application code page is different from the database code page. This is because the server will convert all character data in an SQLDA. To avoid character conversion, you can pass data by defining it in binary string format by using a data type of BLOB or by defining the character data as FOR BIT DATA.

By default, when you invoke DB2 DARI stored procedures and UDFs, they run under a default national language environment which may not match the database's national language environment. Consequently, using country or code page specific operations, such as the C `wchar_t` graphic host variables and functions, may not work as you expect. You need to ensure that, if applicable, the correct environment is initialized when you invoke the stored procedure or UDF.

Package Name Considerations in Mixed Code Page Environments

Package names are determined when you invoke the PRECOMPILE PROGRAM command or API. By default, they are generated based on the first 8-bytes of the application program source file (without the file extension) and are folded to upper case. Optionally, a name can be explicitly defined. Regardless of the origin of a package name, if you are running in an unequal code page environment, the characters for your package names should be in the invariant character set. Otherwise you may experience problems related to the modification of your package name. The database manager will not be able to find the package for the application or a client-side tool will not display the right name for your package.

A package name modification due to character conversion will occur if any of the characters in the package name, or fragments of a character in a multi-byte code page, are not directly mapped to a valid character in the database code page. In such cases, a substitution character replaces the character that is not converted. After such a modification, the package name, when converted back to the application code page, may not match the original package name. An example of a case where this behavior is undesirable is when you use the DB2 Database Director to list and work with packages. Package names displayed may not match the expected names.

To avoid conversion problems with package names, ensure that only characters are used which are valid under both the application and database code pages.

Precompiling and Binding

At precompile/bind time, the precompiler is the executing application. The active code page when the database connection was made prior to the precompile request is used for precompiled statements, and any character data returned in the SQLCA.

Executing an Application

At execution time, the active code page of the user application when a database connection is made is in effect for the duration of the connection. All data is interpreted based on this code page; this includes dynamic SQL statements, user input data, user output data, and character fields in the SQLCA.

A Note of Caution

Failure to follow these guidelines may produce unpredictable results. These conditions cannot be detected by the database manager, so no error or warning message will result. For example, a C application contains the following SQL statements operating against a table T1 with one column defined as C1 CHAR(20):

```
     (0)  EXEC SQL CONNECT TO GLOBALDB;
     (1)  EXEC SQL INSERT INTO T1 VALUES ('a-constant');
          strcpy(sqlstmt, "SELECT C1 FROM T1 WHERE C1='a-constant');
     (2)  EXEC SQL PREPARE S1 FROM :sqlstmt;
Where:
    application code page at bind time = x
    application code page at execution time = y
    database code page = z
```

At bind time, '*a-constant*' in statement (1) is converted from code page **x** to code page **z**. This conversion can be noted as (x→z).

At execution time, '*a-constant*' (x→z) is inserted into the table when statement (1) is executed. However, the WHERE clause of statement (2) will be executed with '*a-constant*' (y→z). If the code points in the constant are such that the two conversions (x→z and y→z) yield different results, the SELECT in statement (2) will fail to retrieve the data inserted by statement (1).

Conversion Between Different Code Pages

Ideally, for optimal performance, your applications should always use the same code page as your database. However, this is not always practical or possible. The DB2 products provide support for character conversion that allows your application and database to use different code pages. Characters from one code page must be mapped to the other code page in order to maintain meaning of the data.

When Does Character Conversion Occur?: Character conversion can occur in the following situations:

- When a client or application accessing a database is running in a code page that is different from the code page of the database.

  This database conversion will occur on the database server machine for both conversions from the application code page to the database code page and from the database code page to the application code page.

  You can minimize or eliminate client/server character conversion in some situations. For example, you could:

  - Create a Windows NT database using code page 850 to match an OS/2 and Windows client application environment that predominately uses code page 850,

    If a Windows ODBC application is used with the IBM DB2 ODBC driver in Windows database client, this problem may be alleviated by the use of the TRANSLATEDLL and TRANSLATEOPTION keywords in the `odbc.ini` or `db2cli.ini` file.

  - Create a DB2 for AIX database using code page 850 to match an OS/2 and DOS client application environment that predominately uses code page 850.

  Note: The DB2 for OS/2 Version 1.0 or Version 1.2 database server does not support character conversion between different code pages. Ensure that the code pages on server and client are compatible. See Table 30 for the code page conversons that are supported.

- When a client or application importing a PC/IXF file runs in a code page that is different from the file being imported.

  This data conversion will occur on the database client machine before the client accesses the database server. Additional data conversion may take place if the application is running in a code page that is different from the code page of the database (as stated in the previous point).

  Data conversion, if any, also depends on how the import utility was called. See the *Administration Guide* for more information.

- When DB2 Connect is used to access data on a DRDA server. In this case character conversion occurs by the receiver of data, as defined by the DRDA rules. For example, data that is sent to DB2 for MVS/ESA is converted to the appropriate MVS coded character set identifier (CCSID) by DB2 for MVS/ESA. The data sent back to the DB2 Connect machine from DB2 for MVS/ESA is converted by DB2 Connect. For more information, see the *DB2 Connect Enterprise Edition Quick Beginnings* for your platform.

Character conversion will **not** occur for:

- File names. You should either use the ASCII invariant set for file names or provide the file name in the hexadecimal values that are physically stored in the file system. Note that if you include a file name as part of a SQL statement, it gets converted as part of the statement conversion.

- Data that is targeted for or comes from a column assigned the FOR BIT DATA attribute, or data used in an SQL operation whose result is FOR BIT or BLOB data. In these cases, the data is treated as a byte stream and no conversion occurs.[6]

  See the *SQL Reference* for unequal code page rules for assigning, comparing, and combining strings.

[6] However, a literal inserted into a column defined as FOR BIT DATA could be converted if that literal was part of an SQL statement which was converted.

- Access to a DB2 for OS/2 Version 1.0 or Version 1.2 database server.
- A DB2 product or platform that does not support, or that does not have support installed, for the desired combination of code pages. In this case, an SQLCODE -332 (SQLSTATE 57017) is returned when you try to run your application.

Character Substitutions During Conversions: When your application converts from one code page to another, it is possible that one or more characters are not represented in the target code page. If this occurs, DB2 inserts a *substitution* character into the target string in place of the character that has no representation. The replacement character is then considered a valid part of the string. In situations where a substitution occurs, the SQLWARN10 indicator in the SQLCA is set to ‘W’.

Note: Any character conversions resulting from using the WCHARTYPE CONVERT precompiler option will not flag a warning if any substitutions take place.

Supported Character Conversions: When data conversion occurs, conversion will take place from a **source code page** to a **target code page**.

The source code page is determined from the source of the data; data from the application has a source code page equal to the application code page, and data from the database has a source code page equal to the database code page.

The determination of target code page is more involved; where the data is to be placed, including rules for intermediate operations, is considered:

- If the data is moved directly from an application into a database, with no intervening operations, the target code page is the database code page.
- If the data is being imported into a database from a PC/IXF file, there are two character conversion steps:

  1. From the PC/IXF file code page (source code page) to the application code page (target code page)
  2. From the application code page (source code page) to the database code page (target code page).

  Exercise caution in situations where two conversion steps might occur. Make sure you follow the supported character conversions listed in Table 30 to avoid a possible loss of character data. Additionally, within each group, only characters which exist in both the source and target code page have meaningful conversions. Other characters are used as “substitutions” and are only useful for converting from the target code page back to the source code page (and may provide meaningless conversions in the second stage).
- If the data is derived from operations performed on character data, where the source may be any of the application code page, the database code page, FOR BIT DATA, or for BLOB data, data conversion is based on a set of rules. Some or all of the data items may have to be converted to an intermediate result, before the final target code page can be determined. See the *SQL Reference* for a summary of these rules and for specific application with individual operators and predicates.

Table 30 on page 553 shows the code page conversions that are supported. Any code page can be converted to any other code page that is listed in the same IBM-defined language group. For example, code page 437 can be converted to 37, 819, 850, 1051, 1252, or 1275.

Note: Character string conversions between multi-byte code pages, for example DBCS and EUC, may result in either an increase or a decrease in the length of the string.

Character Conversion Expansion Factor: When your application successfully completes an attempt to connect to a DB2 database server, you should consider the following fields in the returned SQLCA:

- The second token in the SQLERRMC field (tokens are separated by `X'FF'`) indicates the code page of the database. The ninth token in the SQLERRMC field indicates the code page of the application. Querying the application's code page and comparing it to the database's code page informs the application whether it has established a connection which will undergo character conversions.
- The first and second entries in the SQLERRD array. SQLERRD(1) contains an integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the database code page from the application code page. SQLERRD(2) contains an integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. Refer to the *SQL Reference* for details on using the CONNECT statement.

The considerations for graphic string data should not be a factor in unequal code page situations. Each string always has the same number of characters, regardless of whether the data is in the application or the database code page.

See "Unequal Code Page Situations" on page 382 for information on dealing with unequal code page situations.

Japanese and Traditional-Chinese EUC Code Set Considerations

Extended UNIX Code (EUC) denotes a set of general encoding rules that can support from one to four character sets in UNIX-based operating environments. The encoding rules are based on the ISO 2022 definition for encoding 7-bit and 8-bit data in which control characters are used to separate some of the character sets. EUC is a means of specifying a collection of code sets rather than a code set encoding scheme. A code set based on EUC conforms to the EUC encoding rules but also identifies the specific character sets associated with the specific instances. For example, the IBM-eucJP code set for Japanese refers to the encoding of the Japanese Industrial Standard characters according to the EUC encoding rules. For a list of code pages which are supported, refer to your platform's *Quick Beginnings* book.

Database and client application support for graphic (pure double-byte character) data, while running under EUC code pages with character encoding that is greater than two bytes in length is limited. The DB2 Universal Database products implement strict rules for graphic data that require all characters to be exactly two bytes wide. These rules do not allow many characters from both the Japanese and Traditional-Chinese EUC code pages. To overcome this situation, support is provided at both the application level and the database level to represent Japanese and Traditional-Chinese EUC graphic data using another encoding scheme.

A database created under either Japanese or Traditional-Chinese EUC code pages will actually store and manipulate graphic data using the ISO 10646 UCS-2 code set, a double-byte encoding scheme which is a proper subset of the full ISO 10646 Unicode standard. Similarly, an application running under those code pages will send graphic data to the database server as UCS-2 encoded data. With this support, applications running under EUC code pages can access the same types of data as those running under DBCS code pages. For additional information regarding EUC environments, refer to the *SQL Reference*. The IBM-defined code page identifier associated with UCS-2 encoded data for the DB2 common server products is 13488. The support for UCS-2 encoded data is at Level 1 of the standard.

The ISO 10646 standard specifies the encoding of a number of *combining* characters that are necessary in several scripts, such as Indic, Thai, Arabic and Hebrew. These characters can also be used for a productive generation of characters in Latin, Cyrillic, and Greek scripts. However their presence creates a possibility of an alternative coding for the same text. Although the coding is unambiguous and data integrity is preserved, a processing of text that contains combining characters is more complex. To provide for conformance of applications that choose not to deal with the combining characters, ISO 10646 defines three implementation levels:

- Level 1. Does not allow combining characters.
- Level 2. Allows combining marks from Thai, Indic, Hebrew and Arabic scripts.
- Level 3. Allows all combining marks, including the ones for Latin, Cyrillic, and Greek.

For more information on the Unicode standard, see *Unicode 1.0* Volumes 1 and 2 from *Addison-Wesley*.

If you are working with applications or databases using these character sets you may need to consider dealing with UCS-2 encoded data. When converting UCS-2 graphic data to the application's EUC code page, there is the possibility of an increase in the length of data. For details of data expansion, see "Character Conversion Expansion Factor" on page 377. When large amounts of data are being displayed, it may be necessary to allocate buffers, convert, and display the data in a series of fragments.

The following sections discuss how to handle data in this environment. For these sections, the term EUC is used to refer only to Japanese and Traditional-Chinese EUC character sets. Note that the discussions do not apply to DB2 Korean or Simplified-Chinese EUC support since graphic data in these character sets is represented using the EUC encoding.

Mixed EUC and Double-Byte Client and Database Considerations

The administration of database objects in mixed EUC and double-byte code page environments is complicated by the possible expansion or contraction in the length of object names as a result of conversions between the client and database code page. In particular, many administrative commands and utilities have documented limits to the lengths of character strings which they may take as input or output parameters. These limits are typically enforced at the client, unless documented otherwise. For example, the limit for a table name is 18 bytes. It is possible that a character string which is 18 bytes under a double-byte code page is larger, say 21 bytes, under an EUC code page. This hypothetical 21-byte table name would be considered invalid by such commands as REORGANIZE TABLE if used as an input parameter despite being valid in the target double-byte database. Similarly, the maximum permitted length of output parameters may be exceeded, after conversion, from the database code page to the application code page. This may cause either a conversion error or output data truncation to occur.

If you expect to use administrative commands and utilities extensively in a mixed EUC and double-byte environment, you should define database objects and their associated data with the possibility of length expansion past the supported limits. Administration of an EUC database from a double-byte client will face less restrictions then administration of a double-byte database from an EUC client. Double-byte character strings will always be equal in length or shorter then the corresponding EUC character string. This will generally lead to less problems caused by enforcing the character string length limits.

Note: In the case of SQL statements, validation of input parameters is not conducted until the entire statement has been converted to the database code page. Thus you can use character strings which may be technically longer then allowed when they represented in the client code page, but which meet length requirements when represented in the database code page.

Considerations for Traditional-Chinese Users

Due to the standards definition for Traditional-Chinese, there is a side effect that you may encounter when you convert some characters between double-byte or EUC code pages and UCS-2. There are 189 characters (consisting of 187 radicals and 2 numbers) that share the same UCS-2 code point, when converted, as another character in the code set. When these characters are converted back to double-byte or EUC, they are converted to the code point of the same character's ideograph, with which it shares the same UCS-2 code point, rather then back to the original code point. When displayed, the character appears the same, but has a different code point. Depending on your application's design, you may have to take this behavior into account.

As an example, consider what happens to code point `A7A1` in EUC code page 964, when it is converted to UCS-2 and then converted back to the original code page, EUC 946:

```
EUC 946                UCS-2                 EUC 946

 A7A1 ─────────┐
               ├──────> 4E01 ──────────────> C4A1
 C4A1 ─────────┘
```

Thus, the original code points,`A7A1` and `C4A1` end up as code point `C4A1` after conversion.

If you require the code page conversion tables for EUC code pages 946 (Traditional-Chinese EUC) or 950 (Traditional-Chinese Big-5) and UCS-2, see the *Product and Service Technical Library Technical* home page at the following URL, `"http://www.software.ibm.com/data/db2/support/servinfo/"`.

Developing Japanese or Traditional-Chinese EUC Applications

When developing EUC applications, you need to consider the following items:

- Graphic Data Handling
- Developing for Mixed Code Set Environments

For additional considerations for stored procedures, refer to "Considerations for Stored Procedures" on page 381. Additional language-specific application development issues are discussed in:

- "Japanese or Traditional-Chinese EUC Considerations" on page 444 (for C and C++).
- "Japanese or Traditional-Chinese EUC Considerations" on page 472 (for COBOL).
- "Japanese or Traditional-Chinese EUC Considerations" on page 488 (for FORTRAN).
- "Japanese or Traditional-Chinese EUC Considerations" on page 505 (for REXX).

Graphic Data Handling

This section discusses EUC application development considerations in order to handle graphic data. This includes handling graphic constants, and handling graphic data in UDFs, stored procedures, DBCLOB files, as well as collation.

Graphic Constants

Graphic constants, or literals, are actually classified as mixed character data as they are part of an SQL statement. Any graphic constants in an SQL statement from a Japanese or Traditional-Chinese EUC client are implicitly converted to the graphic encoding by the database server. You can use graphic literals that are composed of EUC encoded characters in your SQL applications. An EUC database server will convert these literals to the graphic database code set which will be UCS-2. Graphic constants from EUC clients should never contain single-width characters such as CS0 7-bit ASCII characters or Japanese EUC CS2 (Katakana) characters.

For additional information on graphic constants see the section on *Graphic Strings* in the *SQL Reference*.

Considerations for UDFs

UDFs are invoked at the database server and are meant to deal with data encoded in the same code set as the database. In the case of databases running under the Japanese or Traditional-Chinese code set, mixed character data is encoded using the EUC code set under which the database is created. Graphic data is encoded using

UCS-2. This means that UDFs need to recognize and handle graphic data which will be encoded with UCS-2.

For example, you create a UDF called VARCHAR which converts a graphic string to a mixed character string. The VARCHAR function has to convert a graphic string encoded as UCS-2 to an EUC representation if the database is created under the EUC code sets.

Considerations for Stored Procedures

A stored procedure, running under either a Japanese or Traditional-Chinese EUC code set, must be prepared to recognize and handle graphic data encoded using UCS-2. When running these code sets, graphic data received or returned through the stored procedure's input/output SQLDA is encoded using UCS-2.

Considerations for DBCLOB Files

There are two important considerations for DBCLOB files:

- The DBCLOB file data is assumed to be in the EUC code page of the application. For EUC DBCLOB files, data is converted to UCS-2 at the client on read, and from UCS-2 at the client on write.
- The number of bytes read or written at the server, is returned in the data length field of the file reference variable based on the number of UCS-2 encoded characters read from or written to the file. The number of bytes actually read from or written to the file may be larger.

Collation

Graphic data is sorted in binary sequence. Mixed data is sorted in the collating sequence of the database applied on each byte. For a discussion on sorting sequences, see *Assignments and Comparisons* in the *SQL Reference*. Due to the possible difference in the ordering of characters in an EUC code set and a DBCS code set for the same country, different results may be obtained when the same data is sorted in an EUC database and in a DBCS database.

Developing for Mixed Code Set Environments

This section deals with the following considerations related to the increase or decrease in the length of data under certain circumstances, when developing applications in a mixed EUC and DBCS environment:

- Unequal Code Page Situations
- Client-Based Parameter Validation
- Using the DESCRIBE Statement
- Using Fixed or Variable Length Data Types
- Character Conversion String Length Overflow
- Rules for String Conversions
- Character Conversions Past Data Type Limits
- Character Conversions in Stored Procedures

Unequal Code Page Situations

Depending on the character encoding schemes used by the application code page and the database code page, there may or may not be a change in the length of a string as it is converted from the source code page to the target code page. A change in length is usually associated with conversions between multi-byte code pages with different encoding schemes, for example DBCS and EUC.

A possible increase in length is usually more serious than a possible decrease in length since an over-allocation of memory is less problematic than an under-allocation. Application considerations for sending or retrieving data depending on where the possible expansion may occur need to be dealt with separately. It is also important to note the differences between a *best-case* and *worst-case* situation when an expansion or contraction in length is indicated. Positive values, indicating a possible expansion, will give the *worst-case* multiplying factor. For example, a value of `2` for the SQLERRD(1) or SQLERRD(2) field means that a maximum of twice the string length of storage will be required to handle the data after conversion. This is a *worst-case* indicator. In this example *best-case* would be that after conversion, the length remains the same.

Negative values for SQLERRD(1) or SQLERRD(2), indicating a possible contraction, also provide the *worst-case* expansion factor. For example, a value of `-1` means that the maximum storage required is equal to the string length prior to conversion. It is indeed possible that less storage may be required, but practically this is of little use unless the receiving application knows in advance how the source data is structured.

To ensure that you always have sufficient storage allocated to cover the maximum possible expansion after character conversion, you should allocate storage equal to the value `max_target_length` obtained from the following calculation:

1. Determine the expansion factor for the data.

   For data transfer from the application to the database:

   ```
   expansion_factor = ABS[SQLERRD(1)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   For data transfer from the database to the application:

   ```
   expansion_factor = ABS[SQLERRD(2)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   In the above calculations, `ABS` refers to the absolute value.

   The check for `expansion_factor = 0` is necessary because some DB2 Universal Database products return 0 in SQLERRD(1) and SQLERRD(2). These servers do not support code page conversions that result in the expansion or shrinkage of data; this is represented by an expansion factor of 1.

2. Intermediate length calculation.

   ```
   temp_target_length = actual_source_length * expansion_factor
   ```

3. Determine the maximum length for target data type.

Target data type	Maximum length of type (type_maximum_length)
CHAR	254
VARCHAR	4 000
LONG VARCHAR	32 700
CLOB	2 147 483 647

4. Determine the maximum target length.

```
if temp_target_length < actual_source_length          1
   max_target_length = type_maximum_length
else
   if temp_target_length > type_maximum_length        2
      max_target_length = type_maximum_length
   else                                               3
      max_target_length = temp_target_length
```

All the above checks are required to allow for overflow which may occur during the length calculation. The specific checks are:

1 Numeric overflow occurs during the calculation of `temp_target_length` in step 2.

If the result of multiplying two positive values together is greater than the maximum value for the data type, the result *wraps around* and is returned as a value less than the larger of the two values.

For example, the maximum value of a 2-byte signed integer (which is used for the length of non-CLOB data types) is 32 767. If the `actual_source_length` is 25 000 and the expansion factor is 2, then `temp_target_length` is theoretically 50 000. This value is too large for the 2-byte signed integer so it gets wrapped around and is returned as -15 536.

For the CLOB data type, a 4-byte signed integer is used for the length. The maximum value of a 4-byte signed integer is 2 147 483 647.

2 `temp_target_length` is too large for the data type.

The length of a data type cannot exceed the values listed in step 3.

If the conversion requires more space than is available in the data type, it may be possible to use a larger data type to hold the result. For example, if a CHAR(250) value requires 500 bytes to hold the converted string, it will not fit into a CHAR value because the maximum length is 254 bytes. However, it may be possible to use a VARCHAR(500) to hold the result after conversion. See “Character Conversions Past Data Type Limits” on page 388 for more information.

3 `temp_target_length` is the correct length for the result.

Using the SQLERRD(1) and SQLERRD(2) values returned when connecting to the database and the above calculations, you can determine whether the length of a string will possibly increase or decrease as a result of character conversion. In general, a value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. (Note that values of ‘0’ will only come from down-level DB2 Universal Database products. Also, these values are undefined for other database server products. Table 14 on page 384 lists values to expect for various application code page and database code page combinations when using DB2 Universal Database.

Table 14. SQLCA.SQLERRD Settings on CONNECT

Application Code Page	Database Code Page	SQLERRD(1)	SQLERRD(2)
SBCS	SBCS	+1	+1
DBCS	DBCS	+1	+1
eucJP	eucJP	+1	+1
eucJP	DBCS	-1	+2
DBCS	eucJP	+2	-1
eucTW	eucTW	+1	+1
eucTW	DBCS	-1	+2
DBCS	eucTW	+2	-1
eucKO	eucKO	+1	+1
eucKR	DBCS	+1	+1
DBCS	eucKR	+1	+1
eucCN	eucCN	+1	+1
eucCN	DBCS	+1	+1
DBCS	eucCN	+1	+1

Expansion at the Database Server: If the SQLERRD(1) entry indicates an expansion at the database server, your application must consider the possibility that length-dependent character data which is valid at the client will not be valid at the database server once it is converted. For example, DB2 products require that column names be no more than eighteen bytes in length. It is possible that a character string which is eighteen bytes in length encoded under a DBCS code page expands past the eighteen byte limit when it is converted to an EUC code page. This means that there may be activities which are valid when the application code page and the database code page are equal, which are invalid when they are different. Exercise caution when you design EUC and DBCS databases for unequal code page situations.

Expansion at the Application: If the SQLERRD(2) entry indicates an expansion at the client application, your application must consider the possibility that length-dependent character data will expand in length after being converted. For example, a row with a CHAR(18) column is retrieved. Under circumstances where the database and application code pages are equal, the length of the data returned is eighteen bytes. However, In an unequal code page situation eighteen bytes of data encoded under a DBCS code page may expand past eighteen bytes when converted to an EUC code page. Thus, additional storage may have to allocated in order to retrieve the complete string.

Client-Based Parameter Validation

An important side effect of potential character data expansion or contraction between the client and server involves the validation of data passed between the client application and the database server. In an unequal code page situation, it is possible that data determined to be valid at the client is actually invalid at the database server after character conversion. Conversely, data that is invalid at the client, may be valid at the database server after conversion.

Any end-user application or API library has the potential of not being able to handle all possibilities in an unequal code page situation. In addition, while some parameter validation such as string length is performed at the client for commands and APIs, the tokens within SQL statements are not verified until they have been converted to the database's code page. This can lead to situations where it is possible to use an SQL statement in an unequal code page environment to access a database object, such as a table, but it will not be possible to access the same object using a particular command or API.

Consider an application that returns data contained in table provided by an end-user, and checks that the table name is not greater than eighteen bytes long. Now consider the following scenarios for this application:

1. A DBCS database is created. From a DBCS client, a table (t1) is created with a table name which is eighteen bytes long. The table name includes several characters which would be greater than two bytes in length if the string is converted to EUC, resulting in the EUC representation of the table name being a total of twenty bytes in length.
2. An EUC client connects to the DBCS database. It creates a table (t2) with a table name which is fourteen bytes long when encoded as EUC and ten bytes long when converted to DBCS. The table name in the DBCS database is ten bytes. The CREATE TABLE is successful.
3. The EUC client creates a table (t3) with a table name that is nine EUC characters in length (twenty bytes). When this name is converted to DBCS its length shrinks to the eighteen byte limit. The CREATE TABLE is successful.
4. The EUC client invokes the application against the each of the tables (t1, t2, and t3) in the DBCS database, which results in:

Table	**Result**
t1	Displays correct results
t2	Displays correct results
t3	The application considers the table name invalid because it is twenty bytes long.

5. The EUC client is used to query the DBCS database from the CLP. Although the table name is twenty bytes long on the client, the queries are successful because the table name is eighteen bytes long at the server.

Using the DESCRIBE Statement

A DESCRIBE performed against an EUC database will return information about mixed character and GRAPHIC columns based on the definition of these columns in the database. This information is based on code page of the server, before it is converted to the client's code page.

When you perform a DESCRIBE against a select list item which is resolved in the application context (for example `VALUES SUBSTR(?,1,2)`); then for any character or graphic data involved, you should evaluate the returned SQLLEN value along with the returned code page. If the returned code page is the same as the application code page, there is no expansion. If the returned code page is the same as the database code page, expansion is possible. Select list items which are FOR BIT DATA (code page 0), or in the application code page are not converted when returned to the application, therefore there is no expansion or contraction of the reported length.

EUC Application with DBCS Database: If your application's code page is an EUC code page, and it issues a DESCRIBE against a database with a DBCS code page, the information returned for CHAR and GRAPHIC columns is returned in the database context. For example, a CHAR(5) column returned as part of a DESCRIBE has a value of five for the SQLLEN field. In the case of non-EUC data, you allocate five bytes of storage when you fetch the data from this column. With EUC data, this may not be the case. When the code page conversion from DBCS to EUC takes place, there may be an increase in the length of the data due to the different encoding used for characters for CHAR columns. For example, with the Traditional-Chinese character set, the maximum increase is double. That is, the maximum character length in the DBCS encoding is two bytes which may increase to a maximum character length of four bytes in EUC. For the Japanese code set, the maximum increase is also double. Note, however, that while the maximum character length in Japanese DBCS is two bytes, it may increase to a maximum character length in Japanese EUC of three bytes. Although this increase appears to be only by a factor of 1.5, the single-byte Katakana characters in Japanese DBCS are only one byte in length, while they are two bytes in length in Japanese EUC. See “Character Conversion Expansion Factor” on page 377 for more information on determining the maximum size.

Possible changes in data length as a result of character conversions apply only to mixed character data. Graphic character data encoding is always the same length, two bytes, regardless of the encoding scheme. To avoid losing the data, you need to evaluate whether an unequal code page situation exists, and whether or not it is between a EUC application and a DBCS database. You can determine the database code page and the application code page from tokens in the SQLCA returned from a CONNECT statement. For more information, refer to “Deriving Code Page Values” on page 369, or refer to the *SQL Reference*. If such a situation exists, your application needs to allocate additional storage for mixed character data, based on the maximum expansion factor for that encoding scheme.

DBCS Application with EUC Database: If your application code page is a DBCS code page and issues a DESCRIBE against an EUC database, a situation similar to that in “EUC Application with DBCS Database” occurs. However, in this case, your application may require less storage than indicated by the value of the SQLLEN field.

The worst case in this situation is that all of the data is single-byte or double-byte under EUC, meaning that exactly SQLLEN bytes are required under the DBCS encoding scheme. In any other situation, less than SQLLEN bytes are required because a maximum of two bytes are required to store any EUC character.

Using Fixed or Variable Length Data Types

Due to the possible change in length of strings when conversions occur between DBCS and EUC code pages, you should consider not using fixed length data types. Depending on whether you require blank padding, you should consider changing the SQLTYPE from a fixed length character string, to a varying length character string after performing the DESCRIBE. For example, if an EUC to DBCS connection is informed of a maximum expansion factor of two, the application should allocate ten bytes (based on the CHAR(5) example in “EUC Application with DBCS Database” on page 386).

If the SQLTYPE is fixed-length, the EUC application will receive the column as an EUC data stream converted from the DBCS data (which itself may have up to five bytes of trailing blank pads) with further blank padding if the code page conversion does not cause the data element to grow to its maximum size. If the SQLTYPE is varying-length, the original meaning of the content of the CHAR(5) column is preserved, however, the source five bytes may have a target of between five and ten bytes. Similarly, in the case of possible data shrinkage (DBCS application and EUC database), you should consider working with varying-length data types.

Note: It may also be necessary to promote the type. For example, if data originally selectable into a VARCHAR(3000) is expected to grow to a maximum of double its current size, a LONG VARCHAR will need to be used instead since VARCHARs can be defined up to a maximum of only 4000 bytes in length.

An alternative to either allocating extra space or promoting the data type is to select the data in fragments. For example, to select the same VARCHAR(3000) which may be up to 6000 bytes in length after the conversion you could perform two selects, of `SUBSTR(VC3000, 1, LENGTH(VC3000)/2)` and `SUBSTR(VC3000, (LENGTH(VC3000)/2)+1)` separately into 2 VARCHAR(3000) application areas. This method is the only possible solution when the data type is no longer promotable. For example, a CLOB encoded in the Japanese DBCS code page with the maximum length of 2 gigabytes is possibly up to twice that size when encoded in the Japanese EUC code page. This means that the data will have to be broken up into fragments since there is no support for a data type in excess of 2 gigabytes in length.

Character Conversion String Length Overflow

In EUC and DBCS unequal code page environments, situations may occur after conversion takes place, when there is not enough space allocated in a column to accommodate the entire string. In this case, the maximum expansion will be twice the length of the string in bytes. In cases where expansion does exceed the capacity of the column, SQLCODE -334 (SQLSTATE 22524) is returned.

This leads to situations that may not be immediately obvious or previously considered as follows:

- An SQL statement may be no longer than 32 765 bytes in length. If the statement is complex enough or uses enough constants or database object names that may be subject to expansion upon conversion, this limit may be reached earlier than expected.
- SQL identifiers are allowed to expand on conversion up to their maximum lengths which is eight bytes for short identifiers and eighteen bytes for long identifiers.
- Host language identifiers are allowed to expand on conversion up to their maximum length which is 255 bytes.
- When the character fields in the SQLCA structure are converted, they are allowed to expand to no more than their maximum defined lengths.

Rules for String Conversions: If you are designing applications for mixed code page environments, familiarize yourself with the concepts in the *Rules for String Conversions* section of the *SQL Reference* for any of the following situations:

- Corresponding string columns in full selects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE statement
- Arguments of the scalar function COALESCE (and VALUE)
- Expression values of the IN list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In these situations, conversions may take place to the application code page instead of the database code page.

Character Conversions Past Data Type Limits: In EUC and DBCS unequal code page environments, situations may occur after conversion takes place, when the length of the mixed character or graphic string exceeds the maximum length allowed for that data type. If the length of the string, after expansion, exceeds the limit of the data type, then type promotion does not occur. Instead, an error message is returned indicating that the maximum allowed expansion length has been exceeded. This situation is more likely to occur while evaluating predicates than with inserts. With inserts, the column width is more readily known by the application, and the maximum expansion factor can be readily taken into account. In many cases, this side effect of character conversion can be avoided by casting the value to an associated data type with a longer maximum length. For example, the maximum length of a CHAR value is 254 bytes while the maximum length of a VARCHAR is 4000 bytes. In cases where expansion does exceed the maximum length of the data type, an SQLCODE -334 (SQLSTATE 22524) is returned.

Character Conversions in Stored Procedures: Mixed character or graphic data specified in host variables and SQLDAs in `sqleproc()` or SQL CALL invocations are converted in situations where the application and database code pages are different. In cases where string length expansion occurs as a result of conversion, you receive an SQLCODE -334 (SQLSTATE 22524) if there is not enough space allocated to handle the expansion. Thus you must be sure to provide enough space for potentially expanding strings when developing stored procedures. You should use varying length data types with enough space allocated to allow for expansion.

Considerations for Distributed Unit of Work (DUOW)

This section describes how your applications can work with remote databases and how they can work with more than one database at a time. Included in the discussion are:

- Remote Unit of Work
- Distributed Unit of Work

With DB2, you can run remote server functions such as BACKUP, RESTORE, DROP DATABASE, CREATE DATABASE and so on as if they were local applications. For more information on using these functions remotely, see the *Administration Guide*.

Remote Unit of Work

A unit of work is a single logical transaction. It consists of a sequence of SQL statements in which either all of the operations are successfully performed or the sequence as a whole is considered unsuccessful.

A remote unit of work lets a user or application program read or update data at one location per unit of work. It supports access to one database within a unit of work. While an application program can access several remote databases, it can only access one database within a unit of work.

A remote unit of work has the following characteristics:

- Multiple requests per unit of work are supported.
- Multiple cursors per unit of work are supported.
- Each unit of work can access only one database.
- The application program either commits or rolls back the unit of work. In certain error circumstances, the server may roll back the unit of work.

Distributed Unit of Work

Distributed unit of work (DUOW) allows an application to access more than one database within a unit of work; that is, the application can switch between databases before committing the data. This gives an application programmer the ability to do work involving multiple databases, local and remote, at the same time.

You can use DUOW to read and update multiple DB2 Universal Database databases within a unit of work. If you have installed DB2 Connect for OS/2 or DB2 Connect for AIX (DDCS Version 2.3.1 or later) you can also use DUOW with other DRDA application servers, such as DB2 for MVS/ESA and DB2 for AS/400. Certain restrictions apply when you use DUOW with other application servers, as described in “Distributed Unit of Work with DB2 Connect” on page 550.

A transaction manager coordinates the commit among multiple databases. If you use a transaction processing (TP) monitor environment such as CICS for AIX , the TP monitor uses its own transaction manager. Otherwise, the transaction manager supplied with

DB2 is used. The DB2 Universal Database transaction manager is an XA (extended architecture) compliant resource manager. The TP monitors access data using the XA interface. Also note that the DB2 Universal Database transaction manager *is not* an XA compliant transaction manager, meaning the transaction manager can only coordinate DB2 databases.

For detailed information about DUOW, see the *Administration Guide*.

When to Use DUOW

DUOW is most useful when you want to work with two or more databases and maintain data integrity. For example, if each branch of a bank has its own database, a money transfer application could do the following:

- Connect to the sender's database
- Read the sender's account balance and verify that enough money is present.
- Reduce the sender's account balance by the transfer amount.
- Connect to the recipient's database
- Increase the recipient's account balance by the transfer amount.
- Commit the databases.

By doing this within one unit of work, you ensure that either both databases are updated or neither database is updated.

Coding SQL for a DUOW Application

Table 15 illustrates how you code SQL statements for DUOW. The left column shows SQL statements that do not use DUOW; the right column shows similar statements with DUOW.

Table 15. RUOW and DUOW SQL Statements

RUOW Statements	**DUOW Statements**
`CONNECT TO D1`	`CONNECT TO D1`
`  SELECT`	`  SELECT`
`  UPDATE`	`  UPDATE`
`  COMMIT`	
	`CONNECT TO D2`
`CONNECT TO D2`	`  INSERT`
`  INSERT`	`  RELEASE CURRENT`
`  COMMIT`	
	`SET CONNECTION D1`
`CONNECT TO D1`	`  SELECT`
`  SELECT`	`  RELEASE D1`
`  COMMIT`	`  COMMIT`
`CONNECT RESET`	

The SQL statements in the left column access only one database for each unit of work. This is a remote unit of work (RUOW) application.

The SQL statements in the right column access more than one database within a unit of work. This is a distributed unit of work (DUOW) application.

Some SQL statements are coded and interpreted differently in a DUOW application:

- The current unit of work does not need to be committed or rolled back before you connect to another database.
- When you connect to another database, the current connection is not disconnected. Instead, it is put into a *dormant* state. If the CONNECT statement fails, the current connection is not affected.
- You cannot connect with the USER/USING clause if a current or dormant connection to the database already exists.
- You can use the SET CONNECTION statement to change a dormant connection to the current connection.

  You can also accomplish the same thing by issuing a CONNECT statement to the dormant database. This is not allowed if you set SQLRULES to STD. You can set the value of SQLRULES using a precompiler option or the SET CLIENT command or API. The default value of SQLRULES (DB2) allows you to switch connections using the CONNECT statement.
- In a select, the cursor position is not affected if you switch to another database and then back to the original database.
- The CONNECT RESET statement does not disconnect the current connection and does not implicitly commit the current unit of work. Instead, it is equivalent to explicitly connecting to the default database (if one has been defined). If an implicit connection is not defined, SQLCODE -1024 (SQLSTATE 08003) is returned.
- You can use the RELEASE statement to mark a connection for disconnection at the next COMMIT. The RELEASE CURRENT statement applies to the current connection, the RELEASE *connection* applies to the named connection, and the RELEASE ALL statement applies to all connections.

  A connection that is marked for release can still be used until it is dropped at the next COMMIT statement. A rollback does not drop the connection; this allows a retry with the connections still in place. Use the DISCONNECT statement (or precompiler option) to drop connections after a commit or rollback.
- The COMMIT statement commits all databases in the unit of work (current or dormant).
- The ROLLBACK statement rolls back all databases in the unit of work, and closes held cursors for all databases whether or not they are accessed in the unit of work.
- All connections (including dormant connections and connections marked for release) are disconnected when the application process terminates.
- Upon any successful connection (including a CONNECT statement with no options, which only queries the current connection) a number will be returned in the SQLERRD(3) and SQLERRD(4) fields of the SQLCA.

  The SQLERRD(3) field returns information on whether the database connected is currently updatable in a unit of work. Its possible values are:

  1 Updatable.
  2 Read-only.

The SQLERRD(4) field returns the following information on the current characteristics of the connection:

0 Not applicable. This state is only possible if running from a down level client which uses one phase commit and is an updater.

1 One-phase commit.

2 One-phase commit (read-only). This state is only applicable to DRDA1 databases in a TP Monitor environment.

3 Two-phase commit.

If you are writing tools or utilities, you may want to issue a message to your users if the connection is read-only.

Precompiling a DUOW Application

The following precompiler options are used when you precompile a DUOW application that does not use an external transaction manager as supplied in a TP Monitor environment. In a TP Monitor environment, these are ignored. Note that when you are precompiling a DUOW application, you should set the CLP connection to a type 1 connection, otherwise you will receive an SQLCODE 30090 (SQLSTATE 25000) when you attempt to precompile your application. For information on setting the connection type, see the *Command Reference*. To precompile your DUOW application, specify the CONNECT 2 precompiler option:

CONNECT (1 | 2)
Specify CONNECT 2 to indicate that this application uses the SQL syntax for DUOW applications, as described in “Coding SQL for a DUOW Application” on page 390. The default, CONNECT 1, means that the normal (RUOW) rules for SQL syntax apply to the application.

SYNCPOINT (ONEPHASE | TWOPHASE | NONE)
Specifies whether updates can be performed on one database per unit of work (the default), multiple databases with two-phase commit (TWOPHASE), or multiple databases without two-phase commit (NONE). Note that you require that a TM_DATABASE be defined for a TWOPHASE commit, except if you are using a transaction manager. For information on how these SYNCPOINT options impact the way your program operates, refer to the concepts section of the *SQL Reference*.

SQLRULES (DB2 | STD)
Specifies whether DB2 rules or standard (STD) rules based on ISO/ANS SQL92 should be used in DUOW applications. DB2 rules allow you to issue a CONNECT statement to a dormant database; STD rules do not allow this.

DISCONNECT (EXPLICIT | CONDITIONAL | AUTOMATIC)
Specifies which database connections are disconnected at COMMIT: only databases that are marked for release with a RELEASE statement (EXPLICIT), all databases that have no open WITH HOLD cursors (CONDITIONAL), or all connections (AUTOMATIC).

For a more detailed description of these precompiler options, see the *Command Reference*.

DUOW precompiler options become effective when the first database connection is made. You can use the SET CLIENT API to supersede connection settings when there are no existing connections (before any connection is established or after all connections are disconnected). You can use the QUERY CLIENT API to query the current connection settings of the application process.

The binder fails if an object referenced in your application program does not exist. There are three possible ways to deal with DUOW applications:

- You can split the application into several files, each of which accesses only one database. You then prep and bind each file against the one database that it accesses.
- You can ensure that each table exists in each database. For example, the branches of a bank might have databases whose tables are identical (except for the data).
- You can use only dynamic SQL.

Specifying Configuration Parameters for a DUOW Application

The following configuration parameters are used for DUOW applications:

TM_DATABASE
Specifies which database will act as a transaction manager for two-phase commit transactions.

RESYNC_INTERVAL
Specifies the number of seconds that the system waits between attempts to try to resynchronize an indoubt transaction. (An indoubt transaction is a transaction that successfully completes the first phase of a two-phase commit but fails during the second phase.)

LOCKTIMEOUT
Specifies the number of seconds before a lock wait will time-out and roll back the current transaction.

TP_MON_NAME
Specifies the name of the TP monitor, if any.

SPM_RESYNC_AGENT_LIMIT
Specifies the number of simultaneous agents that can perform resync operations with the DRDA server using SNA.

SPM_NAME
Identifies the SNA Logical Unit (LU) to be used for two phase commit with a DRDA Server.

SPM_LOG_SIZE
The number of 4 Kbyte pages of each primary and secondary log file used by the SPM to record information on connections, status of current connections, and so on.

For a more detailed description of these configuration parameters, see the *Administration Guide*.

DUOW Restrictions

The following restrictions apply to DUOW in DB2:

- In a transaction processing (TP) Monitor environment such as CICS/6000, the DISCONNECT statement is not supported. If you use DISCONNECT with a TP monitor, you will receive SQLCODE -30090 (SQLSTATE 25000). Instead of DISCONNECT, use RELEASE followed by COMMIT.
- Dynamic COMMIT and ROLLBACK are not supported in a connect type 2 environment. If you use a COMMIT in this environment, it is rejected with SQLCODE -925 (SQLSTATE 2D521). If you use a ROLLBACK in this environment, it is rejected with SQLCODE -926 (SQLSTATE 2D521).
- The precompiler option DISCONNECT CONDITIONAL cannot be used for connections to Version 1 databases. Connections to Version 1 databases are disconnected on COMMIT even if held-cursors are open.
- Although cursors declared WITH HOLD are supported with DUOW, in order for DISCONNECT to succeed, all cursors declared WITH HOLD must be closed and a COMMIT issued before the DISCONNECT request.
- When the services of TP Monitor Environments are used for transaction management, the DUOW options are implicitly CONNECT Type 2, SYNCPOINT TWOPHASE, SQLRULES DB2, DISCONNECT EXPLICIT. Changing these options with precompilation or the SET CLIENT API is not necessary and will be ignored.
- Your application receives an SQLCODE -30090 (SQLSTATE 25000) if it uses the following APIs in a DUOW (CONNECT Type 2), as these APIs are not supported in a DUOW:

  ```
  BACKUP DATABASE
  BIND
  EXPORT
  IMPORT
  LOAD
  MIGRATE DATABASE
  PRECOMPILE PROGRAM
  RESTART DATABASE
  RESTORE DATABASE
  REORGANIZE TABLE
  ROLLFORWARD DATABASE
  ```

- Stored procedures are supported within a DUOW. However, a stored procedure that issues a COMMIT and ROLLBACK statement in a DUOW (CONNECT Type 2) receives an SQLCODE -30090 (SQLSTATE 25000) as these statements are not supported in a DUOW.

Accessing DRDA Servers

If you want to develop applications that can access (or update) different database systems, you should:

1. Use SQL statements and precompile/bind options that are supported on all of the database systems that your applications will access. For example, stored procedures are not supported on all platforms.

   For IBM products, consult Volume 2 of the *IBM SQL Reference, Version 2* (SC26-8416), **before** you start coding.

2. Where possible, have your applications check the SQLSTATE rather than the SQLCODE.

   If your applications will use DDCS and you want to use SQLCODEs, consider using the mapping facility provided by DDCS to map SQLCODE conversions between unlike databases.

3. Test your application with the DRDA databases (such as DB2 for MVS, OS/400, or SQL/DS) that you intend to support. See the *DB2 Connect User's Guide*.

For more information accessing DRDA database systems, see Appendix C, " Programming in a DRDA Environment" on page 541.

Multiple Thread Database Access

One feature of some operating systems is the ability to run several threads of execution within a single process. This allows an application to handle asynchronous events, and makes it easier to create event-driven applications without resorting to polling schemes. This section discusses how the database manager works with multiple threads, and lists some design guidelines that you should keep in mind. See the *DB2 SDK Building Applications* book for your platform to determine if it supports the multi-threading feature.

This section assumes that you are familiar with the terms relating to the development of multi-threaded applications (such as critical section and semaphore). If you are not familiar with these terms, consult the programming documentation for your operating system.

A DB2 application can execute SQL statements from multiple threads using *contexts*. A context is the environment from which an application runs all SQL statements and API calls. All connections, units of work, and other database resources are associated with a specific context. Each context is associated with one or more threads within an application.

For each executable SQL statement in a context, the first run-time services call always tries to obtain a latch. If it is successful, it continues processing. If not (because an SQL statement in another thread of the same context already has the latch), the call is blocked on a signaling semaphore until that semaphore is posted, at which point the call gets the latch and continues processing. The latch is held until the SQL statement has completed processing, at which time it is released by the last run-time services call that was generated for that particular SQL statement.

The net result is that each SQL statement within a context is executed as an atomic unit, even though other threads may also be trying to execute SQL statements at the same time. This action ensures that internal data structures are not altered by different threads at the same time. APIs also use the latch used by run-time services; therefore, APIs have the same restrictions as run-time services routines within each context.

By default, all applications have a single context that is used for all database access. While this is perfect for a single threaded application, the serialization of SQL statements makes a single context inadequate for a multithreaded application. By using the following DB2 APIs, your application can attach a separate context to each thread and allow contexts to be passed between threads:

- `sqleSetTypeCtx()`
- `sqleBeginCtx()`
- `sqleEndCtx()`
- `sqleAttachToCtx()`
- `sqleDetachFromCtx()`
- `sqleGetCurrentCtx()`
- `sqleInterruptCtx()`

Contexts may be exchanged between threads in a process, but not exchanged between processes. One use of multiple contexts is to provide support for concurrent transactions. For the details of how to use these context APIs, refer to the *API Reference* and “Concurrent Transactions” on page 398.

Recommendations for Using Multiple Threads

Follow these guidelines when accessing a database from multiple thread applications:

- **Serialize alteration of data structures.**

  Applications must ensure that user-defined data structures used by SQL statements and database manager routines are not altered by one thread while an SQL statement or database manager routine is being processed in another thread. For example, do not allow a thread to reallocate an SQLDA while it was being used by an SQL statement in another thread.

- **Consider using separate data structures.**

  It may be easier to give each thread its own user-defined data structures to avoid having to serialize their usage. This is especially true for the SQLCA, which is used not only by every executable SQL statement, but also by all of the database manager routines. There are three alternatives for avoiding this problem with the SQLCA:

  1. Use EXEC SQL INCLUDE SQLCA, but add `struct sqlca sqlca` at the beginning of any routine which is used by any thread other than the first thread.

2. Place EXEC SQL INCLUDE SQLCA inside each routine that contains SQL, instead of in the global scope.
3. Replace EXEC SQL INCLUDE SQLCA with `#include "sqlca.h"` and then add `"struct sqlca sqlca"` at the beginning of any routine that uses SQL.

Potential Pitfalls when Using Multiple Threads

An application that uses multiple threads is, understandably, more complex than a single-threaded application. This extra complexity can potentially lead to some unexpected problems. When writing a multithreaded application, exercise caution with the following:

- **Database dependencies between two or more contexts.**

  Each context in an application has its own set of database resources, including locks on database objects. This makes it possible for two contexts, if they are accessing the same database object, to deadlock. The database manager will detect the deadlock and one of the contexts will receive SQLCODE -911 and its unit of work will be rolled back.

- **Application dependencies between two or more contexts.**

  Be careful with any programming techniques that establish inter-context dependencies. Latches, semaphores, and critical sections are examples of programming techniques that can establish such dependencies. If an application has two contexts that have both application and database dependencies between the contexts, it is possible for the application to become deadlocked. If some of the dependencies are outside of the database manager, the deadlock is not detected, thus the application gets suspended or hung.

  As an example of this sort of problem, consider an application that has two contexts, both of which access a common data structure. To avoid problems where both contexts change the data structure simultaneously, the data structure is protected by a semaphore. The contexts look like this:

  ```
  context 1
  SELECT * FROM TAB1 FOR UPDATE....
  UPDATE TAB1 SET....
  get semaphore
  access data structure
  release semaphore
  COMMIT

  context 2
  get semaphore
  access data structure
  SELECT * FROM TAB1...
  release semaphore
  COMMIT
  ```

  Suppose the first context successfully executes the SELECT and the UPDATE statements while the second context gets the semaphore and accesses the data structure. The first context now tries to get the semaphore, but it cannot because the second context is holding the semaphore. The second context now attempts to read a row from table TAB1, but it stops on a database lock held by the first context. The application is now in a state where context 1 cannot finish before context 2 is done and context 2 is waiting for context 1 to finish. The application is deadlocked, but because the database manager does not know about the semaphore dependency neither context will be rolled back. This leaves the application suspended.

Preventing Deadlocks for Multiple Contexts

Because the database manager cannot detect deadlocks between threads, design and code your application in a way that will prevent deadlocks (or at least allow them to be avoided). In the above example, you can avoid the deadlock in several ways:

- Release all locks held before obtaining the semaphore.

  Change the code for context 1 to perform a commit before it gets the semaphore.
- Do not code SQL statements inside a section protected by semaphores.

  Change the code for context 2 to release the semaphore before doing the SELECT.
- Code all SQL statements within semaphores.

  Change the code for context 1 to obtain the semaphore before running the SELECT statement. While this technique will work, it is not highly recommended because the semaphores will serialize access to the database manager, which potentially negates the benefits of using multiple threads.
- Set the LOCKTIMEOUT database configuration parameter to a value other than -1.

  While this will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. When handling the roll back error, context 2 should release the semaphore. Once the semaphore has been released, context 1 can continue and context 2 is free to retry its work .

The techniques for avoiding deadlocks are shown in terms of the above example, but you can apply them to all multithreaded applications. In general, treat the database manager as you would treat any protected resource and you should not run into problems with multithreaded applications.

Concurrent Transactions

Sometimes it is useful for an application to have multiple independent connections called *concurrent transactions*. Using concurrent transactions, an application can connect to several databases at the same time, and can establish several distinct connections to the same database.

The context APIs described in “Multiple Thread Database Access” on page 395 allow an application to use concurrent transactions. Each context created in an application is independent from the other contexts. This means you create a context, connect to a database using the context, and run SQL statements against the database without being affected by the activities such as running COMMIT or ROLLBACK statements of other contexts.

For example, suppose you are creating an application that allows a user to run SQL statements against one database, and keeps a log of the activities performed in a second database. Since the log must be kept up to date, it is necessary to issue a COMMIT statement after each update of the log, but you do not want the user's SQL statements affected by commits for the log. This is a perfect situation for concurrent transactions. In your application, create two contexts: one connects to the user's database and is used for all the user's SQL; the other connects to the log database and is used for updating the log. With this design, when you commit a change to the log database, you do not affect the user's current unit of work.

Another benefit of concurrent transactions is that if the work on the cursors in one connection is rolled back, it has no affect on the cursors in other connections. After the rollback in the one connection, both the work done and the cursor positions are still maintained in the other connections.

Potential Pitfalls when Using Concurrent Transactions

An application that uses concurrent transactions can encounter some problems that cannot arise when writing an application that uses a single connection. When writing an application with concurrent transactions, exercise caution with the following:

- Database dependencies between two or more contexts.

  Each context in an application has its own set of database resources, including locks on database objects. This makes it possible for two contexts, if they are accessing the same database object, to become deadlocked. The database manager will detect the deadlock and one of the contexts will receive an SQLCODE -911 and its unit of work will be rolled back.

- Application dependencies between two or more contexts.

  Switching contexts within a single thread creates dependencies between the contexts. If the contexts also have database dependencies, it is possible for a deadlock to develop. Since some of the dependencies are outside of the database manager, the deadlock will not be detected and the application will be suspended.

  As an example of this sort of problem, consider the following application:

  ```
  context 1
  UPDATE TAB1 SET COL = :new_val

  context 2
  SELECT * FROM TAB1
  COMMIT

  context 1
  COMMIT
  ```

  Suppose the first context successfully executes the UPDATE statement. The update establishes locks on all the rows of TAB1. Now context 2 tries to select all the rows from TAB1. Since the two contexts are independent, context 2 waits on the locks held by context 1. Context 1, however, cannot release its locks until context 2 finishes executing. The application is now deadlocked, but the database manager does not know that context 1 is waiting on context 2 so it will not force one of the contexts to be rolled back. This leaves the application suspended.

Preventing Deadlocks for Concurrent Transactions

Because the database manager cannot detect deadlocks between contexts, you must design and code your application in a way that will prevent deadlocks (or at least avoids deadlocks). In the above example, you can avoid the deadlock in several ways:

- Release all locks held before switching contexts.

  Change the code so that context 1 performs its commit before switching to context 2.

- Do not access a given object from more than one context at a time.

  Change the code so that both the update and the select are done from the same context.

- Set the LOCKTIMEOUT database configuration parameter to a value other than -1.

  While this will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. Once context 2 is rolled back, context 1 can continue executing (which releases the locks) and context 2 can retry its work.

The techniques for avoiding deadlocks are shown in terms of the above example, but you can apply them to all applications which use concurrent transactions.

X/Open XA Interface Programming Considerations

The X/Open** XA Interface is an open standard for coordinating changes to multiple resources, while ensuring the integrity of these changes. Software products known as *transaction processing monitors* typically use the XA interface, and since DB2 supports this interface, one or more DB2 databases may be concurrently accessed as resources in such an environment. See the *Administration Guide* for information about the concepts and implementation of the XA interface support provided by the database manager. See the *DB2 SDK Building Applications* for your platform to determine if it supports the X/Open XA Interface.

Special consideration is required by DB2 when operating in a Distributed Transaction Processing (DTP) environment which uses the XA interface because a different model is used for transaction processing as compared to applications running independent of a TP monitor. The characteristics of this transaction processing model are:

1. Multiple types of recoverable resources (such as DB2 databases) can be modified within a transaction.
2. Resources are updated using two-phase commit to ensure the integrity of the transactions being executed.

3. Application programs send requests to commit or rollback a transaction to the TP monitor product rather than to the managers of the resources. For example, in a CICS environment an application would issue `EXEC CICS SYNCPOINT` to commit a transaction, and issuing `EXEC SQL COMMIT` to DB2 would be invalid and unnecessary.
4. Authorization to run transactions is screened by the TP monitor and related software, so resource managers such as DB2 treat the TP monitor as the single authorized user. For example, any use of a CICS transaction must be authenticated by CICS and the access privilege to the database must be granted to CICS and not to the end user who invokes the CICS application.
5. Multiple programs (transactions) are typically queued and executed on an application server (which appears to DB2 to be a single, long-running application program).

Due to the unique nature of this environment, DB2 has special behavior and requirements for applications coded to run in it:

- Multiple databases can be connected to and updated within a unit of work without consideration of distributed unit of work precompiler options or client settings.
- The DISCONNECT statement is disallowed, and will be rejected with SQLCODE -30090 (SQLSTATE 25000) if attempted.
- The RELEASE statement can be used to specify databases connections to release when a transaction is committed, but this is not recommended. If a connection has been released, subsequent transactions should use the SET CONNECTION statement to connect to the database without requiring authorization.
- COMMIT and ROLLBACK statements are not allowed within stored procedures accessed by a TP monitor transaction.
- When two-phase commit flows are explicitly disabled for a transaction (these are called *LOCAL* transactions in XA Interface terminology) only one database can be accessed within that transaction. This database cannot be one that is accessed using the DB2 Sync Point Manager for two-phase commit over DRDA.
- LOCAL transactions should issue SQL COMMIT or SQL ROLLBACK at the end of each transaction, otherwise the transaction will be considered part of the next transaction which is processed.
- Switching between current database connections is done through the use of either SQL CONNECT or SQL SET CONNECTION. The authorization used for a connection cannot be changed by specifying a userid or password on the CONNECT statement.
- If a database object such as a table, view, or index is not fully qualified in a dynamic SQL statement, it will be implicitly qualified with the single authentication ID that the TP monitor is executing under, rather than user’s ID.
- Any use of DB2 COMMIT or ROLLBACK statements for transactions that are not LOCAL will be rejected. The following codes will be returned:
  - SQLCODE -925 (SQLSTATE 2D521) for static COMMIT

  - SQLCODE -926 (SQLSTATE 2D521) for static ROLLBACK
  - SQLCODE -426 (SQLSTATE 2D528) for dynamic COMMIT
  - SQLCODE -427 (SQLSTATE 2D529) for dynamic ROLLBACK

- CLI requests to COMMIT or ROLLBACK are also rejected.
- Handling database-initiated rollback

  In a DTP environment, if an RM has initiated a rollback (for instance, due to a system error or deadlock) to terminate its own branch of a global transaction, it must not process any more requests from the same application process until a transaction manager-initiated syncpoint request occurs. This includes deadlocks that occur within a stored procedure. For the database manager, this means rejecting all subsequent SQL requests with SQLCODE -918 (SQLSTATE 51021) to inform you that you must roll back the global transaction with the transaction manager's syncpoint service such as using the CICS SYNCPOINT ROLLBACK command in a CICS environment. If for some reason you request the TM to commit the transaction instead, the RM will inform the TM about the rollback and cause the TM to roll back other RMs anyway.
- Cursors declared WITH HOLD

  Cursors declared WITH HOLD are supported in XA/DTP environments for CICS transaction processing monitors.

  In cases where cursors declared WITH HOLD are not supported, the OPEN statement will be rejected with SQLCODE -30090 (SQLSTATE 25000), reason code 03.

  It is the responsibility of the transactions to ensure that cursors specified to be WITH HOLD are explicitly closed when they are no longer required; otherwise they might be inherited by other transactions, causing conflict or unnecessary use of resources.
- Statements which update or change a database are not allowed against databases which do not support two-phase commit request flows. There are two cases for this:

  1. DRDA-accessed databases in environments in which level 2 of DRDA protocol (DRDA2) is not supportable (see "Distributed Unit of Work with DB2 Connect" on page 550).
  2. Down level DB2 databases which do not support XA two-phase commit flows (that is, DB2 for OS/2 Version 1 and earlier).
- Whether a database supports updates in an XA environment can be determined at run-time by issuing a CONNECT statement. The third SQLERRD token will have the value 1 if the database is updatable, and otherwise will have the value 2.
- When updates are restricted, only the following SQL statements will be allowed:

```
CONNECT
DECLARE
DESCRIBE
EXECUTE IMMEDIATE (where the first token or keyword is SET but
                   not SET CONSTRAINTS)
OPEN CURSOR
FETCH CURSOR
CLOSE CURSOR
PREPARE (where the first token or keyword that is not blank or
         left parenthesis is SET (other than SET CONSTRAINTS),
         SELECT, WITH, or VALUES)
SELECT...INTO
VALUES...INTO
```

Any other attempts will be rejected with SQLCODE -30090 (SQLSTATE 25000).

The PREPARE statement will only be useable to prepare SELECT statements. The EXECUTE IMMEDIATE statement is also allowed to execute SQL SET statements that do not return any output value, such as the SET SQLID statement from DB2 for MVS/ESA.

- API Restrictions

  APIs which internally issue a commit in the database and bypass the two-phase commit process will be rejected with SQLCODE -30090 (SQLSTATE 25000). Refer to “DUOW Restrictions” on page 394 for the for a list of these APIs. These APIs are not supported in a DUOW (Connect Type 2).

- Applications should be single-threaded.

  If you intend to develop a multi-threaded application, you should ensure that only one thread uses SQL, or use a multi-process design instead to avoid interleaving of SQL statements from different threads within the same unit of work. If a transaction manager supports multiple processes or multi-threading, you should configure it to serialize the threads so that one thread will execute to a syncpoint before another one begins. An example is the **XASerialize** option of ***all_operation*** in AIX/CICS. (See the *Administration Guide* for more details about the AIX/CICS XAD file which contains this information.)

Note that the above restrictions apply to applications running in TP monitor environment which uses the XA interface. If DB2 databases are not defined for use with the XA interface, these restrictions do not apply, however it is still necessary to ensure that transactions are coded in a way that will not leave DB2 in a state which will adversely affect the next transaction to be run.

Application Linkage

To produce an executable application, you need to link in the application objects with the language libraries, the operating system libraries, the normal database manager libraries, and the libraries of the TP monitor and transaction manager products.

If you are writing applications on a CICS for AIX* system, see the *CICS for AIX Application Programming Guide,* SC33-0814, for SQL restrictions and sample CICS SQL programs.

Working with Large Volumes of Data Across a Network

You can combine the techniques described in Chapter 5, "Writing Stored Procedures" on page 167, and "Row Blocking" on page 152 to significantly improve the performance of applications which need to pass large amounts of data across a network.

Applications that pass arrays, large amounts of data, or packages of data across the network can pass the data in blocks using the SQLDA data structure or host variables as the transport mechanism. This technique is extremely powerful in host languages that support structures.

Either a client application or a server procedure can pass the data across the network. It can be passed using one of the following data types:

- VARCHAR
- LONG VARCHAR
- CLOB
- BLOB

It can also be passed using one of the following graphic types:

- VARGRAPHIC
- LONG VARGRAPHIC
- DBCLOB

See "Data Types" on page 52 for more information about this topic.

Note: Be sure to consider the possibility of character conversion when using this technique. If you are passing data with one of the character string data types such as VARCHAR, LONG VARCHAR, or CLOB, or graphic data types such as VARGRAPHIC, LONG VARGRAPHIC, OR DBCLOB, and the application code page is not the same as the database code page, any non-character data will be converted as if it were character data. To avoid character conversion, you should pass data in a variable with a data type of BLOB.

See "Conversion Between Different Code Pages" on page 374 for more information about how and when data conversion occurs.

Chapter 10. Programming Considerations in a Partitioned Environment

The following sections describe DB2 application programming considerations in a partitioned environment. For example, if your application accesses DB2 data from more than one database manager partition, you need to consider the information contained herein. If you are unfamiliar with partitioned environments, you should read the *DB2 Extended Enterprise Edition Quick Beginnings* for important conceptual information. In addition, you will find the information on partitioned environments in the *Administration Guide* and the *SQL Reference* beneficial.

The specific topics discussed are:

- Using Buffered Inserts
- Example: Extracting Large Volume of Data (largevol.c)
- Error-Handling Considerations
- Debugging

Using Buffered Inserts

A buffered insert is an insert statement that takes advantage of table queues to buffer the rows being inserted, thereby gaining a significant performance improvement. To use a buffered insert, an application must be prepared or bound with the INSERT BUF option.

Buffered inserts can result in substantial performance improvement in applications that perform inserts. Typically, you can use a buffered insert in applications where a single insert statement (and no other database modification statement) is used within a loop to insert many rows and where the source of the data is a VALUES clause in the INSERT statement. Typically the INSERT statement is referencing one or more host variables which change their values during successive executions of the loop. The VALUES clause can specify a single row or multiple rows.

Typical decision support applications require the loading and periodic insertion of new data. This data could be hundreds of thousands of rows. You can prepare and bind applications to use buffered inserts when loading tables.

To cause an application to use buffered inserts, use the PREP command to process the application program source file, or use the BIND command on the resulting bind file. In both situations, you must specify the INSERT BUF option. For more information about binding an application, see “Binding” on page 26. For more information about preparing an application, see “Creating and Preparing the Source Files” on page 20.

Note: Buffered inserts cause the following to occur:

1. The database manager opens one 4 KB buffer for each node on which the table resides.

2. The INSERT statement with the VALUES clause issued by the application causes the row (or rows) to be placed into the appropriate buffer (or buffers).
3. The database manager returns control to the application.
4. The rows in the buffer are sent to the partition when the buffer becomes full, or an event occurs that causes the rows in a partially filled buffer to be sent. A partially filled buffer is flushed when one of the following occurs:
   - The application issues a COMMIT (implicitly or explicitly through application termination) or ROLLBACK.
   - The application issues another statement that causes a savepoint to be taken. OPEN, FETCH, and CLOSE cursor statements do not cause a savepoint to be taken, nor do they close an open buffered insert.

     The following SQL statements will close an open buffered insert:

     – BEGIN COMPOUND SQL
     – COMMIT
     – DDL
     – DELETE
     – END COMPOUND SQL
     – EXECUTE IMMEDIATE
     – GRANT
     – INSERT
     – PREPARE of the same dynamic statement (by name) doing buffered inserts
     – REDISTRIBUTE NODEGROUP
     – REORG
     – REVOKE
     – ROLLBACK
     – RUNSTATS
     – SELECT INTO
     – UPDATE
     – Execution of any other statement, but not another (looping) execution of the buffered INSERT
     – End of application

     The following APIs will close an open buffered insert:

     – BIND (API)
     – REBIND (API)
     – RUNSTATS (API)
     – REORG (API)
     – REDISTRIBUTE (API)

   In any of these situations where another statement closes the buffered insert, the coordinator node waits until every node receives the buffers and the rows are inserted. It then executes the other statement (the one closing the buffered insert), provided all the rows were successfully inserted. See

"Considerations for Using Buffered Inserts" on page 408 for additional details.

The standard interface in a partitioned environment, (without a buffered insert) loads one row at a time doing the following steps (assuming that the application is running locally on one of the partitions):

1. The coordinator node passes the row to the database manager that is on the same node.
2. The database manager uses indirect hashing to determine the partition where the row should be placed:
   a. The target partition receives the row.
   b. The target partition inserts the row locally.
   c. The target partition sends a response to the coordinator node.
3. The coordinator node receives the response from the target partition.
4. The coordinator node gives the response to the application

   The insertion is not committed until the application issues a COMMIT.
5. Any INSERT statement containing the VALUES clause is a candidate for Buffered Insert, regardless of the number of rows or the type of elements in the rows. That is, the elements can be constants, special registers, host variables, expressions, functions and so on.

For a given INSERT statement with the VALUES clause, the DB2 SQL compiler may not buffer the insert based on semantic, performance, or implementation considerations. If you prepare or bind your application with the INSERT BUF option, ensure that it is not dependent on a buffered insert. This means:

- Errors may be reported asynchronously for buffered inserts, or synchronously for regular inserts. If reported asynchronously, an insert error may be reported on a subsequent insert within the buffer, or on the *other* statement which closes the buffer. The statement that reports the error is not executed. For example, consider using a COMMIT statement to close a buffered insert loop. The commit reports an SQLCODE -803 (SQLSTATE 23505) due to a duplicate key from an earlier insert. In this scenario, the commit is not executed. If you want your application to really commit, for example, some updates that are performed before it enters the buffered insert loop, you must reissue the COMMIT statement.
- Rows inserted may be immediately visible through a SELECT statement using a cursor without a buffered insert. With a buffered insert, the rows will not be immediately visible. Do not write your application to depend on these cursor-selected rows if you precompile or bind it with the INSERT BUF option.

Buffered inserts result in the following performance advantages:

- Only one message is sent from the target partition to the coordinator node for each buffer received by the target partition.
- A buffer can contain a large number of rows, especially if the rows are small.

- Parallel processing occurs as insertions are being done across partitions while the coordinator node is receiving new rows.

An application that is bound with INSERT BUF should be written so that the same INSERT statement with VALUES clause is iterated repeatedly before any statement or API that closes a buffered insert is issued.

Note: You should do periodic commits to prevent the buffered inserts from filling the transaction log.

Considerations for Using Buffered Inserts

Buffered inserts exhibit behaviors that can affect an application program. This behavior is caused by the asynchronous nature of the buffered inserts. Based on the values of the row's partitioning key, each inserted row is placed in a buffer destined for the correct partition. These buffers are sent to their destination partitions as they become full, or an event causes them to be flushed. You must be aware of the following, and account for them when designing and coding the application:

- Certain error conditions for inserted rows are not reported when the INSERT statement is executed. They are reported later, when the first statement other than the INSERT (or INSERT to a different table) is executed, such as DELETE, UPDATE, COMMIT, or ROLLBACK. Any statement or API that closes the buffered insert statement can see the error report. Also, any invocation of the insert itself may see an error of a previously inserted row. Moreover, if a buffered insert error is reported by another statement, such as UPDATE or COMMIT, DB2 will not attempt to execute that statement.
- An error detected during the insertion of a *group of rows* causes all the rows of that group to be backed out. A group of rows is defined as all the rows inserted through executions of a buffered insert statement:
  - From the beginning of the unit of work,
  - Since the statement was prepared (if it is dynamic), or
  - Since the previous execution of another updating statement. For a list of statements that close (or flush) a buffered insert, see “Using Buffered Inserts” on page 405.
- An inserted row may not be immediately visible to SELECT statements issued after the INSERT by the same application program, if the SELECT is executed using a cursor.

A buffered INSERT statement is either open or closed. The first invocation of the statement opens the buffered INSERT, the row is added to the appropriate buffer, and control is returned to the application. Subsequent invocations add rows to the buffer, leaving the statement open. While the statement is open, buffers may be sent to their destination partitions, where the rows are inserted into the target table's partition. If any statement or API that closes a buffered insert is invoked while a buffered INSERT statement is open (including invocation of a *different* buffered INSERT statement), or if a PREPARE statement is issued against an open buffered INSERT statement, the open statement is closed before the new request is processed. If the buffered INSERT statement is closed, the remaining buffers are flushed. The rows are then sent to the target partitions and inserted. Only after all the buffers are sent and all the rows are inserted does the new request begin processing.

If errors are detected during the closing of the INSERT statement, the SQLCA for the new request will be filled in describing the error, and the new request is not done. Also, the entire group of rows that were inserted through the buffered INSERT statement *since it was opened* are removed from the database. The state of the application will be as defined for the particular error detected. For example:

- If the error is a deadlock, the transaction is rolled back (including any changes made before the buffered insert section was opened).
- If the error is a unique key violation, the state of the database is the same as before the statement was opened. The transaction remains active, and any changes made before the statement was opened are not affected.

For example, consider the following application that is bound with the buffered insert option:

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
     RESET COUNTER;
  END;
END;
EXEC SQL COMMIT;
```

Suppose the file contains 8 000 values, but value 3 258 is not legal (for example, a unique key violation). Each 1 000 inserts results in the execution of another SQL statement, which then closes the `INSERT INTO t2` statement. During the fourth group of 1 000 inserts, the error for value 3 258 will be detected. It may be detected after the insertion of more values (not necessarily the next one). In this situation, an error code is returned for the `INSERT INTO t2` statement.

The error may also be detected when an insertion is attempted on table `t3`, which closes the `INSERT INTO t2` statement. In this situation, the error code is returned for the `INSERT INTO t3` statement, even though the error applies to table `t2`.

Suppose, instead, that you have 3 900 rows to insert. Before being told of the error on row number 3 258, the application may exit the loop and attempt to issue a COMMIT. The unique-key-violation return code will be issued for the COMMIT statement, and the COMMIT will not be performed. If the application wants to COMMIT the 3000 rows which are in the database thus far (the last execution of `EXEC SQL INSERT INTO t3 ...` ends the savepoint for those 3000 rows), then the COMMIT has to be REISSUED! Similar considerations apply to ROLLBACK as well.

Note: When using buffered inserts, you should carefully monitor the SQLCODES returned to avoid having the table in an indeterminate state. For example, if you remove the `SQLCODE < 0` clause from the THEN DO statement in the above example, the table could end up containing an indeterminate number of rows.

Restrictions on Using Buffered Inserts

The following restrictions apply:

- For an application to take advantage of the buffered inserts, one of the following must be true:
  - The application must either be prepared through PREP or bound with the BIND command and the INSERT BUF option is specified.
  - The application must be bound using the BIND or the PREP API with the SQL_INSERT_BUF option.
- If the INSERT statement with VALUES clause includes long fields or LOBS in the explicit or implicit column list, the INSERT BUF option is ignored for that statement and a normal insert section is done, not a buffered insert. This is not an error condition, and no error or warning message is issued.
- INSERT with fullselect is not affected by INSERT BUF. A buffered INSERT does not improve the performance of this type of INSERT.
- Buffered inserts can be used only in applications, and not through CLP-issued inserts, as these are done through the EXECUTE IMMEDIATE statement.

The application can then be run from any supported client platform.

Example: Extracting Large Volume of Data (largevol.c)

Although DB2 Universal Database provides excellent features for parallel query processing, the single point of connection of an application or an EXPORT command can become a bottleneck if you are extracting large volumes of data. This occurs because the passing of data from the database manager to the application is a CPU-intensive process that executes on a single node (typically a single processor as well).

DB2 Universal Database provides several methods to overcome the bottleneck, so that the volume of extracted data scales linearly per unit of time with an increasing number of processors. The following example, describes the basic idea behind these methods.

Assume that you have a table called EMPLOYEE which is stored on 20 nodes, and you generate a mailing list (FIRSTNME, LASTNAME, JOB) of all employees who are in a legitimate department (that is, WORKDEPT is not NULL).

The following query is run on each node in parallel, and then generates the entire answer set at a single node (the coordinator node):

```
  SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
```

But, the following query could be run on each partition in the database (that is, if there are five partitions, five separate queries are required, one at each partition). Each query generates the set of all the employee names whose record is on the particular partition where the query runs. Each local result set can be redirected to a file. The result sets then need to be merged into a single result set. (On AIX, you can use a property of Network File System (NFS) files to automate that merging. If all the partitions direct their answer sets to the same file on an NFS mount, the results are merged.)

```
  SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
                         AND NODENUMBER(NAME) = CURRENT NODE
```

The result can either be stored in a local file (meaning that the final result would be 20 files, each containing a portion of the complete answer set), or in a single NFS-mounted file.

The following example uses the second method, so that the result is in a single file that is NFS mounted across the 20 nodes. The NFS locking mechanism ensures serialization of writes into the result file from the different partitions . Note that this example, as presented, runs on the AIX platform with an NFS file system installed.

```
#define _POSIX_SOURCE
#define INCL_32

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sqlenv.h>
#include <errno.h>
#include <sys/access.h>
#include <sys/flock.h>
#include <unistd.h>

#define BUF_SIZE 1500000  /* Local buffer to store the fetched records */
#define MAX_RECORD_SIZE 80 /* >= size of one written record */

int main(int argc, char *argv[]) {

    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
       char dbname[10];  /* Database name (argument of the program) */
       char userid[9];
       char passwd[19];
       char first_name[21];
       char last_name[21];
       char job_code[11];
    EXEC SQL END DECLARE SECTION;

       struct flock unlock ;  /* structures and variables for handling */
       struct flock lock ;  /* the NFS locking mechanism */
       int lock_command ;
       int lock_rc ;
       int iFileHandle ;  /* output file */
       int iOpenOptions = 0 ;
       int iPermissions ;
       char * file_buf ;  /* pointer to the buffer where the fetched
                             records are accumulated */
       char * write_ptr ;  /* position where the next record is written */
       int buffer_len = 0 ;  /* length of used portion of the buffer */
```

```
/* Initialization */

   lock.l_type = F_WRLCK;  /* An exclusive write lock request */
   lock.l_start = 0;  /* To lock the entire file */
   lock.l_whence = SEEK_SET;
   lock.l_len = 0;
   unlock.l_type = F_UNLCK;  /* An release lock request */
   unlock.l_start = 0;  /* To unlock the entire file */
   unlock.l_whence = SEEK_SET;
   unlock.l_len = 0;
   lock_command = F_SETLKW;  /* Set the lock */
   iOpenOptions = O_CREAT;  /* Create the file if not exist */
   iOpenOptions |= O_WRONLY;  /* Open for writing only */

/* Connect to the database */

   if (argc == 3) {
      strcpy( dbname, argv[2] ); /* get database name from the argument */
      EXEC SQL CONNECT TO :dbname IN SHARE MODE ;
      if ( SQLCODE != 0 ) {
         printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
                SQLCODE );
        exit(1);
    }
   }
   else if ( argc == 5 ) {
     strcpy( dbname, argv[2] ); /* get  database name from the argument */
   strcpy (userid, argv[3]);
   strcpy (passwd, argv[4]);
   EXEC SQL CONNECT TO :dbname IN SHARE MODE USER :userid USING :passwd;
   if ( SQLCODE != 0 ) {
      printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
              SQLCODE );
        exit( 1 );
     }
   }
   else {
     printf ("\nUSAGE: largevol txt_file database [userid passwd]\n\n");
     exit( 1 ) ;
   } /* endif */

  /* Open the input file with the specified access permissions */

 if ( ( iFileHandle = open(argv[1], iOpenOptions, 0666 ) ) == -1 ) {
   printf( "Error: Could not open %s.\n", argv[2] ) ;
   exit( 2 ) ;
}

/* Set up error and end of table escapes */

EXEC SQL WHENEVER SQLERROR GO TO ext ;
EXEC SQL WHENEVER NOT FOUND GO TO cls ;

/* Declare and open the cursor */

EXEC SQL DECLARE c1 CURSOR FOR
         SELECT firstnme, lastname, job FROM employee
         WHERE workdept IS NOT NULL
```

```
             AND NODENUMBER(lastname) = CURRENT NODE;
    EXEC SQL OPEN c1 ;

    /* Set up the temporary buffer for storing the fetched result */

    if ( ( file_buf = ( char * ) malloc( BUF_SIZE ) ) == NULL ) {
       printf( "Error: Allocation of buffer failed.\n" ) ;
       exit( 3 ) ;
    }
    memset( file_buf, 0, BUF_SIZE ) ; /* reset the buffer */
    buffer_len = 0 ;  /* reset the buffer length */
    write_ptr = file_buf ;  /* reset the write pointer */
    /* For each fetched record perform the following    */
    /*  - insert it into the buffer following the       */
    /*    previously stored record                      */
    /*  - check if there is still enough space in  the  */
    /*    buffer for the next record and lock/write/    */
    /*    unlock the file and initialize the buffer     */
    /*    if not                                        */

    do {
       EXEC SQL FETCH c1 INTO :first_name, :last_name, :job_code;
        buffer_len += sprintf( write_ptr, "%s %s %s\n",
                               first_name, last_name, job_code );
        buffer_len = strlen( file_buf ) ;
       /* Write the content of the buffer to the file if */
       /* the buffer reaches the limit                   */
       if ( buffer_len >= ( BUF_SIZE - MAX_RECORD_SIZE ) ) {
        /*  get excl. write lock */
        lock_rc = fcntl( iFileHandle, lock_command, &lock );
              if ( lock_rc != 0 ) goto file_lock_err;
              /*  position at the end of file */
              lock_rc = lseek( iFileHandle, 0, SEEK_END );
             if ( lock_rc < 0 ) goto file_seek_err;
             /* write the buffer */
             lock_rc = write( iFileHandle,
                               ( void * ) file_buf, buffer_len );
             if ( lock_rc < 0 ) goto file_write_err;
               /* release the lock */
              lock_rc = fcntl( iFileHandle, lock_command, &unlock );
              if ( lock_rc != 0 ) goto file_unlock_err;
              file_buf[0] = '\0' ;  /* reset the buffer */
               buffer_len = 0 ;  /* reset the buffer length */
               write_ptr = file_buf ;  /* reset the write pointer */
       }
       else {
          write_ptr = file_buf + buffer_len ;  /* next write position */
       }
    } while (1) ;

cls:
    /* Write the last piece of data out to the file */
    if (buffer_len > 0) {
       lock_rc = fcntl(iFileHandle, lock_command, &lock);
       if (lock_rc != 0) goto file_lock_err;
      lock_rc = lseek(iFileHandle, 0, SEEK_END);
       if (lock_rc < 0) goto file_seek_err;
       lock_rc = write(iFileHandle, (void *)file_buf, buffer_len);
       if (lock_rc < 0) goto file_write_err;
       lock_rc = fcntl(iFileHandle, lock_command, &unlock);
      if (lock_rc != 0) goto file_unlock_err;
    }
    free(file_buf);
      close(iFileHandle);
    EXEC SQL CLOSE c1;
    exit (0);
 ext:
    if ( SQLCODE != 0 )
       printf( "Error:  SQLCODE = %ld.\n", SQLCODE );
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CONNECT RESET;
    if ( SQLCODE != 0 ) {
       printf( "CONNECT RESET Error:  SQLCODE = %ld\n", SQLCODE );
       exit(4);
    }
    exit (5);
 file_lock_err:
    printf("Error: file lock error = %ld.\n",lock_rc);
    /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(6);
file_seek_err:
    printf("Error: file seek error = %ld.\n",lock_rc);
    fcntl(iFileHandle, lock_command, &unlock);  /* unconditional unlock of the file */
   exit(7);
file_write_err:
    printf("Error: file write error = %ld.\n",lock_rc);
    fcntl(iFileHandle, lock_command, &unlock);  /* unconditional unlock of the file */
    exit(8);
file_unlock_err:
    printf("Error: file unlock error = %ld.\n",lock_rc);
   fcntl(iFileHandle, lock_command, &unlock);  /* unconditional unlock of the file */
    exit(9);
}
```

This method is applicable not only to a select from a single table, but also for more complex queries. If, however, the query requires noncollocated operations (that is, the Explain shows more than one subsection besides the Coordinator subsection), this can result in too many processes on some partitions if the query is run in parallel on all partitions. In this situation, you can store the query result in a temporary table TEMP on as many partitions as required, then do the final extract in parallel from TEMP.

If you want to extract all employees, but only for selected job classifications, you can define the TEMP table with the column names, FIRSTNME, LASTNAME, and JOB, as follows:

```
INSERT INTO TEMP
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
       AND EMPNO NOT IN (SELECT EMPNO FROM EMP_ACT WHERE
                         EMPNO<200)
```

Then you would do the parallel extract on TEMP.

When defining the TEMP table, consider the following:

- If the query specifies an aggregation GROUP BY, you should define the partitioning key of TEMP as a subset of the GROUP BY columns.
- The partitioning key of the TEMP table should have enough cardinality (that is, number of distinct values in the answer set) to ensure that the table is equally distributed across the partitions on which it is defined.
- Use the NOT LOGGED INITIALLY option on the CREATE TABLE TEMP statement.

If you require the final answer set (which is the merged partial answer set from all nodes) to be sorted, you can:

- Specify the SORT BY clause on the final SELECT
- Do an extract into a separate file on each partition
- Merge the separate files into one output set using, for example, the `sort -m` AIX command

Error-Handling Considerations

In a partitioned environment, DB2 breaks up SQL statements into subsections, each of which are processed on the partition that contains the relevant data. As a result, an error may occur on a partition that does not have access to the application. This does not occur in a single-partition environment.

You should consider the following:

- Non-CURSOR (EXECUTE) non-severe errors
- CURSOR non-severe errors
- Severe errors
- Merged multiple SQLCA structures
- How to identify the partition that returned the error

If an application ends abnormally because of a severe error, indoubt transactions may be left in the database. (An indoubt transaction pertains to global transactions when one phase completes successfully, but the system fails before the a subsequent can complete, leaving the database in an inconsistent state.) For information on handling them, see the *Administration Guide*.

Severe Errors

If a severe error occurs in DB2 Universal Database, one of the following will occur:

- The database manager on the node where the error occurs shuts down.

  Active units of work are not rolled back.

  In this situation, you must recover the node and any databases that were active on the node when the shutdown occurred.
- All agents are forced off the database at the node where the error occurred.

  All units of work on that database are rolled back.

In this situation, the database at the node where the error occurred is marked as inconsistent. Any attempt to access it results in either SQLCODE -1034 (SQLSTATE 58031) or SQLCODE -1015 (SQLSTATE 55025) being returned. Before you or any other application on another node can access the database at this node, you must run the RESTART DATABASE command against the database. Refer to the *Command Reference* for information on this command.

The severe error SQLCODE -1224 (SQLSTATE 55032) can occur for a variety of reasons. If you receive this message, check the SQLCA, which will indicate which node failed. Then check the `db2diag.log` file shared between the nodes for details. See "Identifying the Partition that Returned the Error" on page 417 for additional information.

Merged Multiple SQLCA Structures

One SQL statement may be executed by a number of agents on different nodes, and each agent may return a different SQLCA for different errors or warnings. The coordinating agent also has its own SQLCA. In addition, the SQLCA also has fields that indicate global numbers (such as the *sqlerrd* fields that indicate row counts). To provide a consistent view for applications, all the SQLCA values are merged into one structure. This structure is described in *SQL Reference*.

Error reporting is as follows:

- Severe error conditions are always reported. As soon as a severe error is reported, no additions beyond the severe error are added to the SQLCA.
- If no severe error occurs, a deadlock error takes precedence over other errors.
- For all other errors, the SQLCA for the first negative SQLCODE is returned to the application.
- If no negative SQLCODEs are detected, the SQLCA for the first warning (that is, positive SQLCODE) is returned to the application. The exception to this occurs if a data manipulation operation is issued on a table that is empty on one partition, but has data on other partitions. The SQLCODE +100 is only returned to the application if agents from all partitions return SQL0100W, either because the table is empty on all partitions or there are no rows that satisfy the WHERE clause in an UPDATE statement.
- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- The values in the *sqlerrd* fields that indicate row counts are accumulations from all agents.

An application may receive a subsequent error or warning after the problem that caused the first error or warning is corrected. Errors are reported to the SQLCA to ensure that the first error detected is given priority over others. This ensures that an error caused by an earlier error cannot overwrite the original error. Severe errors and deadlock errors are given higher priority because they require immediate action by the coordinating agent.

Identifying the Partition that Returned the Error

If a partition returns an error or warning, its number is in the SQLERRD(6) field of the SQLCA. The number in this field is the same as that specified for the partition in the `db2nodes.cfg` file.

If an SQL statement or API call is successful, the partition number in this field is not significant.

For information about the SQLCA, see the *SQL Reference*.

Debugging

You can use the tools described in the following sections for use in debugging your applications. For more information, refer to the *Troubleshooting Guide*.

Diagnosing a Looping or Suspended application

It is possible that, after you start a query or application, you suspect that it is suspended (it does not show any activity) or that it is looping (it shows activity, but no results are returned to the application). Ensure that you have turned lock timeouts on. In some situations, however, no error is returned. In these situations, you may find the tools described in the *Troubleshooting Guide*, as well as the Database system monitor snapshot helpful.

One of the functions of the database system monitor that is useful for debugging applications is to display the status of all active agents. To obtain the greatest use from a snapshot, ensure that statement collection is being done before you run the application (preferably immediately after you run DB2START) as follows:

```
db2_all "db2 UPDATE MONITOR SWITCHES USING STATEMENT ON"
```

When you suspect that your application or query is either stalled or looping, issue the following command:

```
db2_all "db2 GET SNAPSHOT FOR AGENTS ON database
```

Refer to the *System Monitor Guide and Reference* for information on how read the information collected from the snapshot, and for the details of using the database system monitor.

Chapter 11. Programming in C and C++

This chapter discusses special programming considerations for database manager applications written in C or C++. This includes the following topics:

- Language Restrictions
- Include Files
- Embedding SQL Statements
- Host Variables
- Supported SQL Data Types

Programming Considerations

Special host language programming considerations are discussed in the following pages. Included is information on language restrictions, host-language-specific include files, embedding SQL statements, host variables, and supported data types for host variables.

Language Restrictions

The following sections describe the C/C++ language restrictions.

Macro Expansion

Do not use C preprocessor constructs such as, macros, `#ifdefs`, or symbol substitutions in the host variable declare section or in any SQL statements. You can use these constructs elsewhere in your application. The SQL precompiler does not handle preprocessor functions such as macro processing or symbol substitution, and does not support conditional precompilation (`#ifdefs`). The following example demonstrates incorrect placement of the C preprocessor constructs:

```
#define NAME_LENGTH 20
EXEC SQL BEGIN DECLARE SECTION;
  char name [NAME_LENGTH];    /* incorrect placement of macro */
EXEC SQL END DECLARE SECTION;
#ifdef PRODUCTION   /* careful - both statements will be precompiled */
EXEC SQL SELECT NAME INTO :name FROM PRODUCTION.PAYROLL
  WHERE ID = 10;
#else
EXEC SQL SELECT NAME INTO :name FROM TESTING.PAYROLL
  WHERE ID = 10;
#endif
```

Input and Output Files

By default, the input file can have the following extensions:

.sqc	For C files on either platform.
.sqC	For C++ files on case sensitive platforms (like AIX).
.sqx	For C++ files on case insensitive platforms (like OS/2).

The corresponding default output files have the following extensions:

.c	For C files on any platform.
.C	For C++ files on case sensitive platforms (like AIX).
.cxx	For C++ files on case insensitive platforms (like OS/2).
.cpp	For C++ files for the Borland C++ compiler on OS/2 or Windows 3.1.

You can use the OUTPUT precompile option to override the name and path of the output modified source file. If you use the TARGET C or TARGET CPLUSPLUS precompile option, the input file does not need a particular extension.

Including Files

There are two methods for including files: the EXEC SQL INCLUDE statement and the #include macro. The precompiler will ignore the #include, and only process files included with the EXEC SQL INCLUDE statement.

To locate files included using `EXEC SQL INCLUDE`, the DB2 C/C++ precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll;`

  If the file specified in the INCLUDE statement is not enclosed in quotes, as above, the C/C++ precompiler searches for `payroll.sqc`, then `payroll.h`, in each directory in which it looks. On UNIX operating systems, the C/C++ precompiler searches for `payroll.sqC`, then `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory in which it looks. On OS/2 or Windows-based operating systems, the C/C++ precompiler searches for `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory in which it looks.

- `EXEC SQL INCLUDE 'pay/payroll.h';`

  If the file name is enclosed in quotes, as above, no extension is added to the name.

  If the file name in quotes does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, on UNIX based systems, if DB2INCLUDE is set to '`/disk2:myfiles/c`', the C/C++ precompiler searches for '`./pay/payroll.h`', then '`/disk2/pay/payroll.h`', and finally '`./myfiles/c/pay/payroll.h`'. The path where the file is actually found is displayed in the precompiler messages. On OS/2 and Windows-based operating systems, substitute back slashes (\) for the forward slashes in the above example.

Note: The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

To help relate compiler errors back to the original source the precompiler generates ANSI #line macros in the output file. This allows the compiler to report errors using the file name and line number of the source or included source file, rather than the precompiler output. Some debuggers and other tools that relate source code to object code do not always work well with the #line macro. If the tool you wish to use behaves unexpectedly, use the NOLINEMACRO option (used with db2 PREP) when precompiling. This will prevent the #line macros from being generated.

Trigraph Sequences

Some characters from the C or C++ character set are not available on all keyboards. These characters can be entered into a C or C++ source program using a sequence of three characters called a *trigraph*. Trigraphs are not recognized in SQL statements. The precompiler recognizes the following trigraphs within host variable declarations:

Trigraph	Definition
??(	Left bracket '['
??)	Right bracket ']'
??<	Left brace '{'
??>	Right brace '}'

The remaining trigraphs listed below may occur elsewhere in a C or C++ source program:

Trigraph	Definition
??=	Hash mark '#'
??/	Back slash '\'
??'	Caret '^'
??!	Vertical Bar '|'
??–	Tilde '˜'

C++ Type Decoration Consideration

When writing a stored procedure or a UDF using C++, you may want to consider declaring the procedure or UDF as:

```
extern "C" ...procedure or function declaration...
```

The `extern "C"` prevents type decoration of the function name by the C++ compiler. Without this declaration, you have to include all the type decoration for the function name when you call the stored procedure, or issue the CREATE FUNCTION statement.

Include Files

The host-language-specific include files (header files) for C and C++ have the file extension `.h`. The include files that are intended to be used in your applications, are described below.

SQL (sql.h)
This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLADEF (sqladef.h)
This file contains function prototypes used by precompiled C and C++ applications.

SQLAPREP (sqlaprep.h) This file contains definitions required to write your own precompiler.

SQLCA (sqlca.h) This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

SQLCLI (sqlcli.h) This file contains the function prototypes and constants needed to write a simple Call Level Interface (DB2 CLI) application. The functions in this file are common to both X/Open Call Level Interface and ODBC Core Level.

SQLCLI1 (sqlcli1.h) This file contains the function prototypes and constants needed to write a Call Level Interface (DB2 CLI) that makes use of the more advanced features in DB2 CLI. Many of the functions in this file are common to both X/Open Call Level Interface and ODBC Level 1. In addition, this file also includes X/Open-only functions and DB2-specific functions.

This file includes both `sqlcli.h` and `sqlext.h` (which contains ODBC Level2 API definitions).

SQLCODES (sqlcodes.h) This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqlda.h) This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

SQLEAU (sqleau.h) This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

SQLENV (sqlenv.h) This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLEXT (sqlext.h) This file contains the function prototypes and constants of those ODBC Level 1 and Level 2 APIs that are not part of the X/Open Call Level Interface specification and is therefore used with the permission of Microsoft** Corporation.

SQLE819A (sqle819a.h) If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.h)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.h)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.h)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.h)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.h)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLJACB (sqljacb.h)
This file defines constants, structures and control blocks for the DB2 Connect interface.

SQLMON (sqlmon.h)
This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLSTATE (sqlstate.h)
This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLSYSTM (sqlsystm.h)
This file contains the platform-specific definitions used by the database manager APIs and data structures.

SQLUDF (sqludf.h)
This file defines constants and interface structures for writing User Defined Functions (UDFs). For more information on this file, see “The UDF Include File: sqludf.h” on page 305.

SQLUTIL (sqlutil.h)
This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

SQLUV (sqluv.h) This file defines structures, constants, and prototypes for the asynchronous Read Log API, and APIs used by the table load and unload vendors.

SQLUVEND (sqluvend.h) This file defines structures, constants and prototypes for the APIs to be used by the storage management vendors.

SQLXA (sqlxa.h) This file contains function prototypes and constants used by applications that use the X/Open XA Interface.

Embedding SQL Statements

Embedded SQL statements consist of the following three elements:

Element	**Correct Syntax**
Statement initializer	EXEC SQL
Statement string	Any valid SQL statement
Statement terminator	semicolon (;).

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

The following rules apply to embedded SQL statements:

- You can begin the SQL statement string on the same line as the keyword pair or a separate line. The statement string can be several lines long. Do not split the EXEC SQL keyword pair between lines.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.

  C/C++ comments can be placed after the statement terminator because the precompiler distinguishes them as separate from the SQL statement.
- Multiple SQL statements and C/C++ statements may be placed on the same line. For example:

  ```
  EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
  ```
- The SQL precompiler leaves CR/LFs and TABs in a quoted string as is.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end. Do not place SQL comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the C/C++ language.

  You can use comments in a static statement string wherever blanks are allowed. Use the C/C++ comment delimiters /* */, or the SQL comment symbol (--). //-style C++ comments are not permitted within static SQL statements, but they may be used elsewhere in your program. The precompiler removes comments before processing the SQL statement. You **cannot** use the C and C++ comment delimiters /* */ or // in a dynamic SQL statement. However, you can use them elsewhere in your program.

- You can continue SQL string literals and delimited identifiers over line breaks in C and C++ applications. To do this, use a back slash (\) at the end of the line where the break is desired. For example:

  ```
  EXEC SQL SELECT "NA\
  ME" INTO :n FROM staff WHERE name='Sa\
  nders';
  ```

  Any new line characters (such as carriage return and line feed) are not included in the string or delimited identifier.

- Substitution of white space characters such as end-of-line and TAB characters occur as follows:

  – When they occur outside quotes (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  – When they occur inside quotes, the end-of-line characters disappear, provided the string is continued properly for a C program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

Host Variables

Host variables are C or C++ language variables that are referenced within SQL statements. They allow an application to pass input data to and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other C/C++ variable. Obey the rules described in the following sections when naming, declaring, and using host variables.

Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.

- Begin host variable names with prefixes other than SQL or sql which are reserved for system use. For example:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char  varsql;     /* allowed */
    char  sqlvar;     /* not allowed */
    char  SQL_VAR;    /* not allowed */
  EXEC SQL END DECLARE SECTION;
  ```

- The precompiler considers host variable names as global to a module. This does not mean, however, that host variables have to be declared as global variables; it is perfectly acceptable to declare host variables as local variables within functions. For example, the following code will work correctly:

```
void f1(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_1;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
}
void f2(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
}
```

It is also possible to have several local host variables with the same name as long as they all have the same type and size. To do this, declare the first occurrence of the host variable to the precompiler between BEGIN DECLARE SECTION and END DECLARE SECTION statements, and leave subsequent declarations of the variable out of declare sections. The following code shows an example of this:

```
void f3(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  char host_var_3[25];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
}
void f4(int i)
{
char host_var_3[25];
EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
}
```

Since f3 and f4 are in the same module, and since host_var_3 has the same type and length in both functions, a single declaration to the precompiler is sufficient to use it in both places.

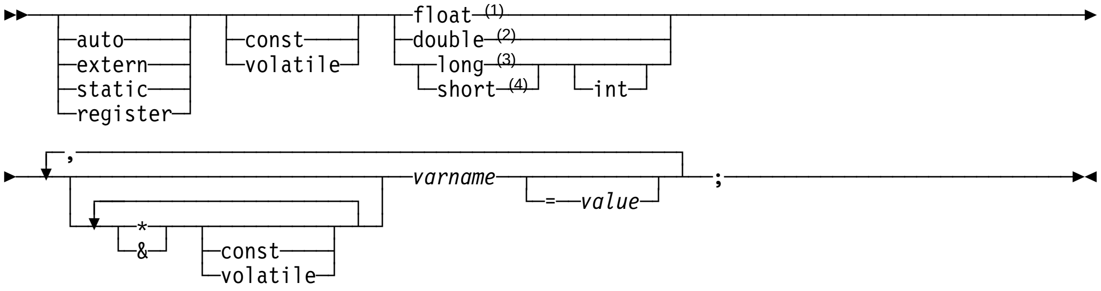
Declaring Host Variables

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The C/C++ precompiler only recognizes a subset of valid C or C++ declarations as valid host variable declarations. These declarations define either numeric or character variables. Using typedefs for host variable types is not allowed, and host variables cannot be embedded in structures. You can declare C++ class data members as host variables. For more information on classes, see “Using Class Data Members as Host Variables” on page 438.

A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The application must ensure that output variables are long enough to contain the values that they receive.

Figure 26 shows the syntax for declaring numeric host variables in C or C++.



Notes:

[1] REAL (SQLTYPE 480), length 4

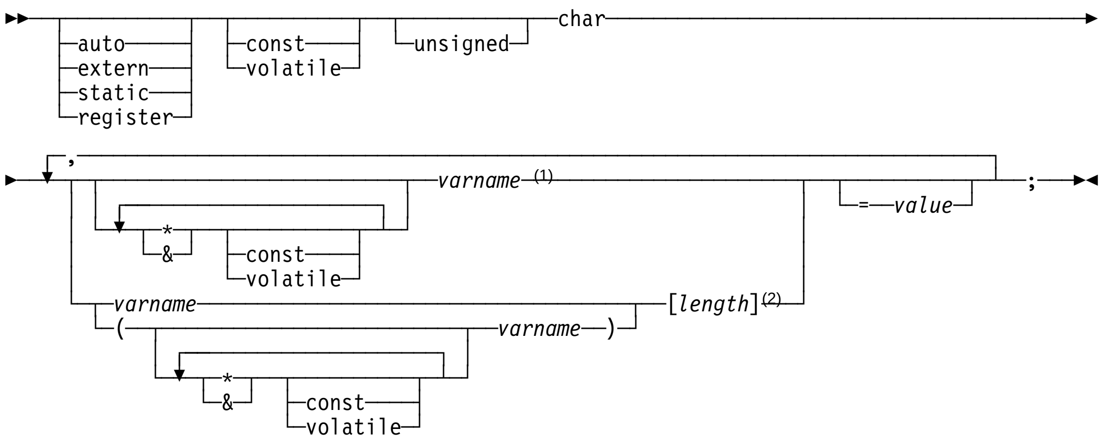
[2] DOUBLE (SQLTYPE 480), length 8

[3] INTEGER (SQLTYPE 496)

[4] SMALLINT (SQLTYPE 500)

Figure 26. Syntax for Numeric Host Variables in C or C++

Figure 27 on page 428 shows the syntax for declaring fixed and null-terminated character host variables in C or C++.



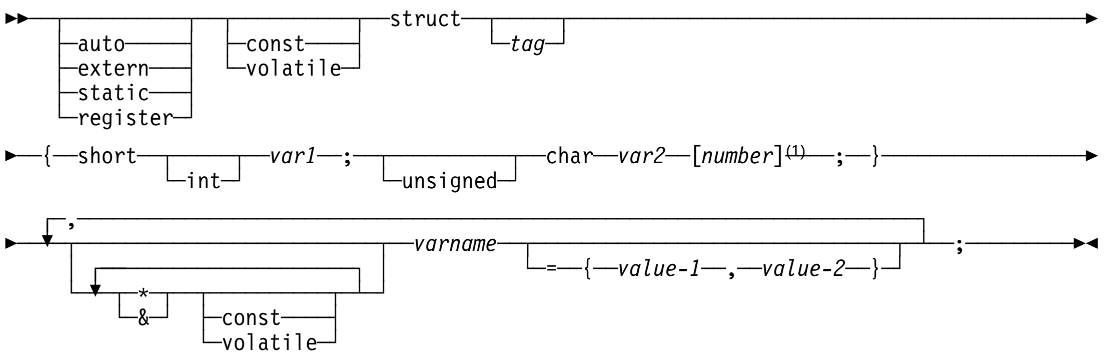
Notes:

[1] CHAR (SQLTYPE 452), Length 1

[2] Null-terminated C string (SQLTYPE 460)

Figure 27. Form 1: Syntax for Fixed and Null-terminated Character Host Variables in C/C++

Figure 28 shows the syntax for declaring variable length character host variables in C or C++.



Note:

[1] In **form 2**, the value of *number* determines if the host variable is VARCHAR (SQLTYPE 448) or LONG VARCHAR (SQLTYPE 456).

Figure 28. Form 2: Syntax for Variable Length Character Host Variables in C/C++

Notes:

1. Although the database manager converts character data to either **form 1** or **form 2** whenever possible, **form 1** corresponds to column types CHAR or VARCHAR while **form 2** corresponds to column types VARCHAR and LONG VARCHAR.
2. If **form 1** is used with a length specifier *[n]*, the length must be no greater than 4000, and the string contained by the variable should be null-terminated.
3. If **form 2** is used, the length specifier must be no greater than 32 700.
4. In **form 2**, *var1* and *var2* must be simple variable references (no operators), and cannot be used as host variables (*varname* is the host variable).
5. *varname* can be a simple variable name or it can include operators, such as **varname*. See “Pointer Data Types” on page 437 for more information.
6. The precompiler determines the SQLTYPE and SQLLEN of all host variables. If a host variable appears in an SQL statement with an indicator variable, the SQLTYPE is assigned to be the base SQLTYPE plus one, for the duration of that statement.
7. The precompiler permits some declarations which are not syntactically valid in C or C++. Refer to your compiler documentation if in doubt of a particular declaration syntax.

Indicator Variables

Indicator variables should be declared as a `short` data type.

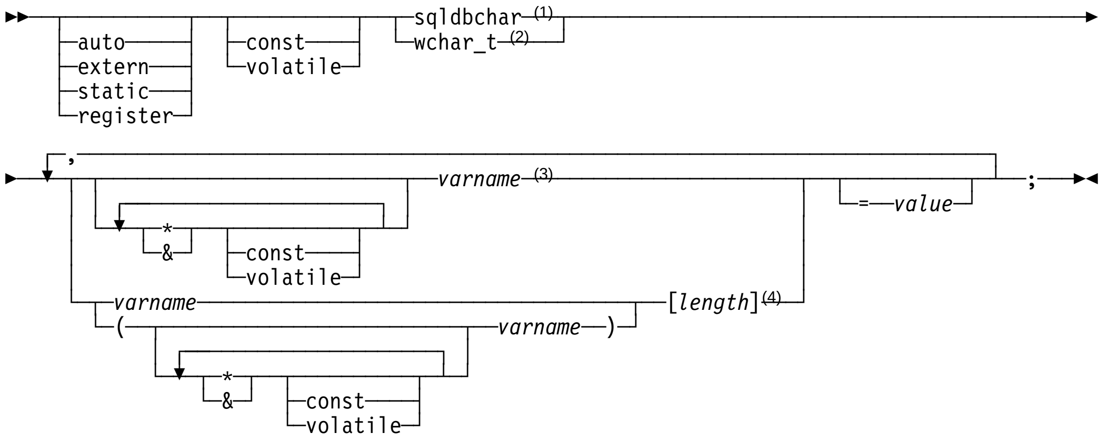
Graphic Host Variable Declarations in C or C++

Graphic host variable declarations can take one of three forms:

- Single-graphic form
- null-terminated graphic form
- VARGRAPHIC structured form

For details on using graphic host variables, see “Handling Graphic Host Variables” on page 440.

Figure 29 on page 430 shows the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.



Notes:

[1] To determine which of the two graphic types should be used, see “Selecting the wchar_t or sqldbchar Data Type” on page 441.

[2] To determine which of the two graphic types should be used, see “Selecting the wchar_t or sqldbchar Data Type” on page 441.

[3] GRAPHIC (SQLTYPE 468), Length 1

[4] Null-terminated graphic string (SQLTYPE 400)

Figure 29. Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)

Notes:

1. The single-graphic form declares a fixed-length graphic string host variable of length 1 with SQLTYPE of 468 or 469.
2. *value* is an initializer. A wide-character string literal (L-literal) should be used if WCHARTYPE CONVERT precompiler option is used.
3. *length* must be a constant that is greater than or equal to 1 and not greater than the maximum length of VARGRAPHIC which is 2000.
4. Null-terminated graphic strings are handled differently depending on the value of the standards level precompile option setting. See “Null-terminated Strings” on page 435 for details.

Figure 30 on page 431 shows the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

```
►►──┬──────────┬──┬──────────┬──struct──┬─────┬──────────────────────────────────►
    ├─auto─────┤  ├─const────┤          └─tag─┘
    ├─extern───┤  └─volatile─┘
    ├─static───┤
    └─register─┘

►──{──short──┬─────┬──var-1──;──┬─sqldbchar─(1)─┬──var-2──[length]──;──}──────────►
             └─int─┘            └─wchar_t─(2)───┘

     ┌─,──────────────────────────────────────────┐
►────▼─┬──────────────────────────────────┬──┤ Variable ├──;─┴──────────────────────►◄
       │  ┌───────────────────────────┐   │
       └──▼─┬─*─┬──┬──────────┬───────┴───┘
            └─&─┘  ├─const────┤
                   └─volatile─┘
```

Variable:

```
├──variable-name──┬────────────────────────────────┬──────────────────────────────┤
                  └─=──{──value-1──,──value-2──}──┘
```

Notes:

[1] To determine which of the two graphic types should be used, see “Selecting the wchar_t or sqldbchar Data Type” on page 441.

[2] To determine which of the two graphic types should be used, see “Selecting the wchar_t or sqldbchar Data Type” on page 441.

Figure 30. Syntax for Graphic Declaration (VARGRAPHIC Structured Form)

Notes:

1. *length* must be a constant that is greater than 1 and not greater than the maximum length of LONG VARGRAPHIC which is 16350.
2. *var-1* and *var-2* must be simple variable references (no operators) and cannot be used as host variables.
3. *value-1* and *value-2* are initializers for *var-1* and *var-2*. *value-1* must be an integer and *value-2* should be a wide-character string literal (L-literal) if WCHARTYPE CONVERT precompiler option is used.
4. The struct *tag* can be used to define other data areas, but these cannot be used as host variables.
5. The value of *length* determines if the host variable is VARGRAPHIC (SQLTYPE 464) or LONG VARGRAPHIC (SQLTYPE 472).

LOB Data Declarations in C or C++

Figure 31 on page 432 shows the syntax for declaring large object (LOB) host variables in C or C++.

```
►►──┬──────────┬──┬──────────┬──SQL TYPE IS──┬─BLOB───┬──(length──┬───┬──)──────────────►
    ├─auto─────┤  ├─const────┤               ├─CLOB───┤           ├─K─┤
    ├─extern───┤  └─volatile─┘               └─DBCLOB─┘           ├─M─┤
    ├─static───┤                                                  └─G─┘
    └─register─┘

     ┌─,───────────────────────────────────────────────────────────────────────────┐
►────▼─┬────────────────────────────┬──variable-name──┬──────────────────────────────────┬─┴──;──────►◄
       │  ┌──────────────────────┐  │                 ├─={init-len,"init-data"}──────────┤
       └──▼─┬─*─┬──┬──────────┬──┴──┘                 ├─=SQL_BLOB_INIT("init-data")──────┤
            └─&─┘  ├─const────┤                       ├─=SQL_CLOB_INIT("init-data")──────┤
                   └─volatile─┘                       └─=SQL_DBCLOB_INIT("init-data")────┘
```

Figure 31. Syntax for Large Object (LOB) Host Variables in C/C++

Notes:

1. The SQL TYPE IS clause is needed in order to distinguish the three LOB-types from each other so that type-checking and function resolution can be carried out for LOB-type host variables that are passed to functions.
2. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
3. For DBCLOB 1 <= lob-length <= 1 073 741 823.
4. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G may be in mixed case.
5. The maximum length allowed for the initialization string, *"init-data"*, is 4000 bytes (the same as the existing limit on C/C++ strings within the precompiler).
6. The initialization length, *init-len*, must be a numeric constant (i.e. it cannot include K, M, or G).
7. A length for the LOB must be specified; that is, the following declaration is not permitted:

   ```
   SQL TYPE IS BLOB my_blob;
   ```

8. If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code.
9. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

   Note: Wide character literals, for example, `L"Hello"`, should only be used in a precompiled program if the WCHARTYPE CONVERT precompile option is selected.

10. The precompiler generates a structure tag which can be used to cast to the host variable's type.

BLOB Example:

Declaration:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

Results in the generation of the following structure:

```
static struct my_blob_t {
      unsigned long    length;
      char             data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

CLOB Example:

Declaration:

```
 volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

Results in the generation of the following structure:

```
 volatile struct var1_t {
      unsigned long    length;
      char             data[131072000];
 } * var1, var2 = {10, "data5data5"};
```

DBCLOB Example:

Declaration:

```
 SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

Precompiled with the WCHARTYPE NOCONVERT option, results in the generation of the following structure:

```
struct my_dbclob1_t {
     unsigned long    length;
     sqldbchar        data[30000];
} my_dbclob1;
```

Declaration:

```
 SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
     unsigned long    length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

LOB Locator Declarations in C or C++

Figure 32 on page 434 shows the syntax for declaring large object (LOB) locator host variables in C or C++.

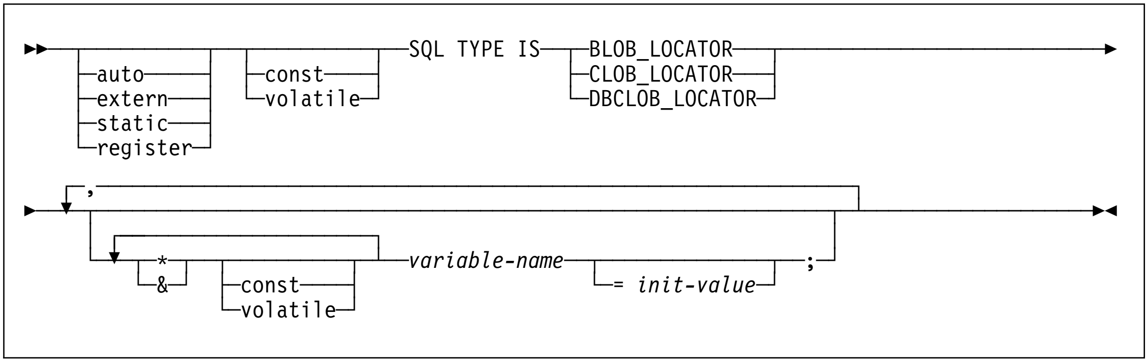


Figure 32. Syntax for Large Object (LOB) Locator Host Variables in C/C++

Notes:

1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR may be in mixed case.
2. *init-value* permits the initialization of pointer and reference locator variables. Other types of initialization will have no meaning.

CLOB Locator Example (other LOB locator type declarations are similar):

Declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

Results in the generation of the following declaration:

```
long my_locator;
```

File Reference Declarations in C or C++

Figure 33 shows the syntax for declaring file reference host variables in C or C++.

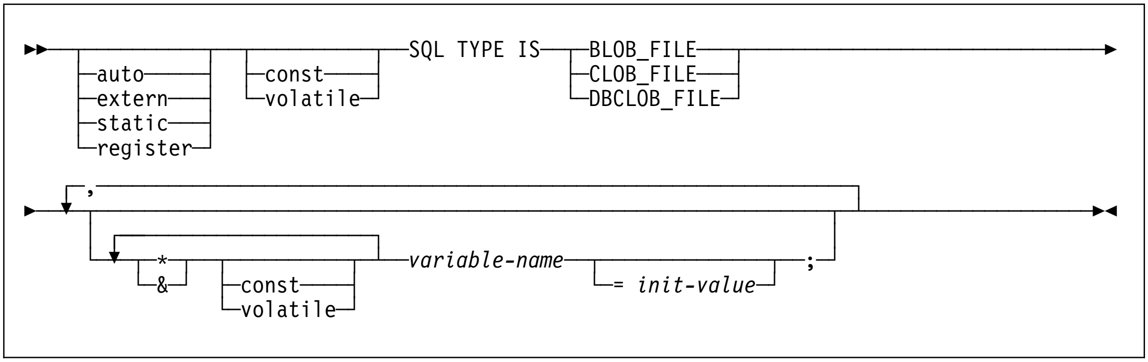


Figure 33. Syntax for File Reference Host Variables in C/C++

Note:

- SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE may be in mixed case.

CLOB File Reference Example (other LOB file reference type declarations are similar):

Declaration:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

Results in the generation of the following structure:

```
static volatile struct {
     unsigned long    name_length;
     unsigned long    data_length;
     unsigned long    file_options;
              char    name[255];
} my_file;
```

Initializing Host Variables

In C++ declare sections, you cannot initialize host variables using parentheses. The following example shows the correct and incorrect methods of initialization in a declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
  short my_short_2 = 5;       /* correct   */
  short my_short_1(5);        /* incorrect */
EXEC SQL END DECLARE SECTION;
```

Null-terminated Strings

C/C++ null-terminated strings have their own SQLTYPE (460/461 for character and 468/469 for graphic).

C/C++ null-terminated strings are handled differently depending on the value of the LANGLEVEL precompiler option. If a host variable of one of these SQLTYPEs and declared length *n* is specified within an SQL statement, and the number of bytes (for character types) or double-byte characters (for graphic types) of data is *k*, then:

- If the LANGLEVEL option on the PREP command is SAA1 (the default):

  For Output:

If...	**Then...**
k > *n*	*n* characters are moved to the target host variable, SQLWARN1 is set to 'W', SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to *k*.

$k = n$	*k* characters are moved to the target host variable and SQLWARN1 is set to `'N'`, and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.
$k < n$	*k* characters are moved to the target host variable and a null character is placed in character $k + 1$. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

For Input:

When the database manager encounters an input host variable of one of these SQLTYPEs that does not end with a null-terminator, it will assume that character $n+1$ will contain the null-terminator character.

- If the LANGLEVEL option on the PREP command is MIA:

For Output:

If...	**Then...**
$k >= n$	$n - 1$ characters are moved to the target host variable, SQLWARN1 is set to `'W'`, and SQLCODE 0 (SQLSTATE 01501). The *n*th character is set to the null-terminator. If an indicator variable was specified with the host variable, the value of the indicator variable is set to *k*.
$k + 1 = n$	*k* characters are moved to the target host variable, and the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.
$k + 1 < n$	*k* characters are moved to the target host variable, $n - k -1$ blanks are appended on the right starting at character $k + 1$, then the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

For Input:

When the database manager encounters an input host variable of one of these SQLTYPEs that does not end with a null character, SQLCODE -302 (SQLSTATE 22501) are returned

When specified in any other SQL context, a host variable of SQLTYPE 460 with length *n* is treated as a VARCHAR data type with length *n* as defined above. When specified in any other SQL context, a host variable of SQLTYPE 468 with length *n* is treated as a VARGRAPHIC data type with length *n* as defined above.

Pointer Data Types

Host variables may be declared as pointers to specific data types with the following restrictions:

- If a host variable is declared as a pointer, then no other host variable may be declared with that same name within the same source file. The following example is not allowed:

  ```
  char mystring[20];
  char (*mystring)[20];
  ```

- Use parentheses when declaring a pointer to a null-terminated character array. In all other cases, parentheses are not allowed. For example:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char  (*arr)[10];  /* correct   */
    char  *(arr);      /* incorrect */
    char  *arr[10];    /* incorrect */
  EXEC SQL END DECLARE SECTION;
  ```

  The first declaration is a pointer to a 10-byte character array. This is a valid host variable. The second is an invalid declaration. The parentheses are not allowed in a pointer to a character. The third declaration is an array of pointers. This is not a supported data type.

  The host variable declaration:

  ```
    char *ptr
  ```

  is accepted, but it does not mean *null-terminated character string of undetermined length*. Instead, it means a *pointer to a fixed-length, single character host variable*. This may not be what is intended. To define a pointer host variable that can indicate different character strings, use the first declaration form above.

- When pointer host variables are used in SQL statements, they should be prefixed by the same number of asterisks as they were declared with, as in the following example:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char (*mychar)[20];    /* Pointer to character array of 20 bytes */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL SELECT column INTO :*mychar;   /* Correct */
  ```

- Only the asterisk may be used as an operator over a host variable name.

- The maximum length of a host variable name is not affected by the number of asterisks specified, because asterisks are not considered part of the name.

- Whenever using a pointer variable in an SQL statement, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager.

Using Class Data Members as Host Variables

You can declare class data members as host variables (but not classes or objects themselves). The following example illustrates the method to use:

```
class STAFF
{
    private:
         EXEC SQL BEGIN DECLARE SECTION;
           char       staff_name[20];
           short int   staff_id;
           double     staff_salary;
         EXEC SQL END DECLARE SECTION;
         short      staff_in_db;
    .
    .
};
```

Figure 34. Example of Declaring Class Data Members as Host Variables

Data members are only directly accessible in SQL statements through the implicit *this* pointer provided by the C++ compiler in class member functions. You **cannot** explicitly qualify an object instance (such as `SELECT name INTO :my_obj.staff_name ...`) in an SQL statement.

If you directly refer to class data members in SQL statements, the database manager resolves the reference using the *this* pointer. For this reason, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager. (This is true whenever pointer host variables are involved in SQL statements.)

The following example shows how you might directly use class data members which you have declared as host variables, in an SQL statement.

```
 class STAFF
 {
     .
     .
     public:
     .
     .
        short int hire( void )
        {
          EXEC SQL INSERT INTO staff ( name,id,salary )
            VALUES ( :staff_name, :staff_id, :staff_salary );
          staff_in_db = (sqlca.sqlcode == 0);
          return sqlca.sqlcode;
        }
};
```

Figure 35. Example of Using Class Data Members Directly in an SQL Statement

In this example, class data members `staff_name`, `staff_id`, and `staff_salary`, are used directly in the INSERT statement. Because they have been declared as host variables (see the example in Figure 34 on page 438), they are implicitly qualified to the current object with the *this* pointer. In SQL statements, you can also refer to data members that are not accessible through the *this* pointer. You do this by referring to them indirectly using pointer or reference host variables.

The following example shows a new method, *asWellPaidAs* that takes a second object, *otherGuy*. This method references its members indirectly through a local pointer or reference host variable, as you cannot reference its members directly within the SQL statement.

```
short int STAFF::asWellPaidAs( STAFF otherGuy )
{
    EXEC SQL BEGIN DECLARE SECTION;
      short &otherID = otherGuy.staff_id
      double otherSalary;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL SELECT SALARY INTO :otherSalary
      FROM STAFF WHERE id = :otherID;
      if( sqlca.sqlcode == 0 )
         return staff_salary >= otherSalary;
      else
         return 0;
}
```

Figure 36. Example of Using Class Data Members Indirectly in an SQL Statement

Using Qualification and Member Operators

You **cannot** use the C++ scope resolution operator '::', nor the C/C++ member operators '.' or '->' in embedded SQL statements. You can easily accomplish the same thing through use of local pointer or reference variables, which are set outside the SQL statement to point to the desired scoped variable, and then used inside the SQL statement to refer to it. The following example shows the correct method to use:

```
EXEC SQL BEGIN DECLARE SECTION;
  char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
  SELECT name INTO :localName FROM STAFF
  WHERE name = 'Sanders';
```

Handling Graphic Host Variables

To handle graphic data in C or C++ applications, use host variables based on either the `wchar_t` C/C++ data type or the `sqldbchar` data type provided by DB2. You can assign these types of host variables to columns of a table that are GRAPHIC, VARGRAPHIC, or DBCLOB. For example, you can update or select DBCS data from GRAPHIC or VARGRAPHIC columns of a table.

There are three valid forms for a graphic host variable:

- Single-graphic form.

  Single-graphic host variables have an SQLTYPE of 468/469 that is equivalent to GRAPHIC(1) SQL data type. (See Figure 29 on page 430.)

- Null-terminated graphic form.

  Null-terminated refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). They have an SQLTYPE of 400/401. (See Figure 29 on page 430.)

- VARGRAPHIC structured form.

  VARGRAPHIC structured host variables have an SQLTYPE of 464/465 if their length is between 1 and 2 000 bytes. They have an SQLTYPE of 472/473 if their length is between 2000 and 16 350 bytes. (SeeFigure 30 on page 431 .)

Multi-byte Character Encoding: Some character encoding schemes, particularly those from east Asian countries require multiple bytes to represent a character. This external representation of data is called the *multi-byte character code* representation of a character and includes double-byte characters (characters represented by two bytes). Graphic data in DB2 consists of double-byte characters.

To manipulate character strings with double-byte characters, it may be convenient for an application to use an internal representation of data. This internal representation is called the *wide-character code* representation of the double-byte characters and is the format customarily used in the `wchar_t` C/C++ data type. There are ANSI C and X/OPEN Portability Guide 4 (XPG4) conformant subroutines available to process wide-character data and to convert data in wide-character format to and from multi-byte format.

Note that although an application can process character data in either multi-byte format or wide-character format, interaction with the database manager is done with DBCS (multi-byte) character codes only. That is, data is stored in and retrieved from GRAPHIC columns in DBCS format. The WCHARTYPE precompiler option is provided to allow application data in wide-character format to be converted to/from multi-byte format when it is exchanged with the database engine.

Selecting the wchar_t or sqldbchar Data Type: While the size and encoding of DB2 graphic data is constant from one platform to another for a particular code page, the size and internal format of the ANSI C or C++ `wchar_t` data type depends on which compiler you use and which platform you are on. The `sqldbchar` data type, however, is defined by DB2 to be two bytes in size, and is intended to be a portable way of manipulating DBCS and UCS-2 data in the same format in which it is stored in the database. For more information on UCS-2 data, see "Japanese and Traditional-Chinese EUC Code Set Considerations" on page 377.

You can define all DB2 C graphic host variable types using either `wchar_t` or `sqldbchar`. You must use `wchar_t` if you build your application using the WCHARTYPE CONVERT precompile option (as described in "The WCHARTYPE Precompiler Option"). If you build your application with the WCHARTYPE NOCONVERT precompile option, you should use `sqldbchar` for maximum portability between different DB2 client and server platforms. You may use `wchar_t` with WCHARTYPE NOCONVERT, but only on platforms where `wchar_t` is defined as two bytes in length.

If you incorrectly use either `wchar_t` or `sqldbchar` in host variable declarations, you will receive an SQLCODE 15 (no SQLSTATE) at precompile time.

The WCHARTYPE Precompiler Option: Using the WCHARTYPE precompiler option, you can specify which graphic character format you want to use in your C/C++ application. This option provides you with the flexibility to choose between having your graphic data in multi-byte format or in wide-character format. There are two possible values for the WCHARTYPE option:

CONVERT

If you select the WCHARTYPE CONVERT option, character codes are converted between the graphic host variable and the database manager. For graphic input host variables, the character code conversion from wide-character format to multi-byte DBCS character format is performed before the data is sent to the database manager, using the ANSI C function `wcstombs()`. For graphic output host variables, the character code conversion from multi-byte DBCS character format to wide-character format is performed before the data received from the database manager is stored in the host variable, using the ANSI C function `mbstowcs()`.

The advantage to using WCHARTYPE CONVERT is that it allows your application to fully exploit the ANSI C mechanisms for dealing with wide-character strings (L-literals, 'wc' string functions, etc.) without having to explicitly convert the data to multi-byte format before communicating with the database manager. The disadvantage is that the implicit conversions may have an impact on the performance of your application at run time, and may increase memory requirements.

If you select WCHARTYPE CONVERT, declare all graphic host variables using `wchar_t` instead of `sqldbchar`.

If you want WCHARTYPE CONVERT behavior, but your application does not need to be precompiled (for example, a CLI application), then define the C preprocessor macro `SQL_WCHART_CONVERT` at compile time. This ensures that certain definitions in the DB2 header files use the data type `wchar_t` instead of `sqldbchar`.

Note: The WCHARTYPE CONVERT precompile option is not currently supported in programs running on the DB2 Windows 3.1 client. For those programs, use the default (WCHARTYPE NOCONVERT).

NOCONVERT (default)
If you choose the WCHARTYPE NOCONVERT option, or do not specify any WCHARTYPE option, no implicit character code conversion occurs between the application and the database manager. Data in a graphic host variable is sent to and received from the database manager as unaltered DBCS characters. This has the advantage of improved performance, but the disadvantage that your application must either refrain from using wide-character data in `wchar_t` host variables, or must explicitly call the `wcstombs()` and `mbstowcs()` functions to convert the data to and from multi-byte format when interfacing with the database manager.

If you select WCHARTYPE NOCONVERT, declare all graphic host variables using the `sqldbchar` type for maximum portability to other DB2 client/server platforms.

See the *Command Reference* for more information.

Other guidelines you need to observe are:

- Since `wchar_t` or `sqldbchar` support is used to handle DBCS data, its use requires DBCS or EUC capable hardware and software. This support is only available in the DBCS environment of DB2 Universal Database.
- Non-DBCS characters, and wide-characters which can be converted to non-DBCS characters, should not be used in graphic strings. *Non-DBCS characters* refers to single-byte characters, and non-double byte characters. Graphic strings are not validated to ensure that their values contain only double-byte character code points. Graphic host variables must contain only DBCS data, or, if WCHARTYPE CONVERT is in effect, wide-character data which converts to DBCS data. You should store mixed double-byte and single-byte data in character host variables.

Note that mixed data host variables are unaffected by the setting of the WCHARTYPE option.

- In applications where the WCHARTYPE NOCONVERT precompile option is used, L-literals should not be used in conjunction with graphic host variables, since L-literals are in wide-character format. An L-literal is a C wide-character string literal prefixed by the letter L which has the data type `"array of wchar_t"`. For example, `L"dbcs-string"` is an L-literal.
- In applications where the WCHARTYPE CONVERT precompile option is used, L-literals can be used to initialize `wchar_t` host variables, but cannot be used in SQL statements. Instead of using L-literals, SQL statements should use graphic string constants, which are independent of the WCHARTYPE setting.
- The setting of the WCHARTYPE option affects graphic data passed to and from the database manager using the SQLDA structure as well as host variables. If WCHARTYPE CONVERT is in effect, graphic data received from the application through an SQLDA will be presumed to be in wide-character format, and will be converted to DBCS format via an implicit call to `wcstombs()`. Similarly, graphic output data received by an application will have been converted to wide-character format before being placed in application storage.
- Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. Ordinary fenced stored procedures may be precompiled with either the CONVERT or NOCONVERT options, which will affect the format of graphic data manipulated by SQL statements contained in the stored procedure. In either case, however, any graphic data passed into the stored procedure through the SQLDA will be in DBCS format. Likewise, data passed out of the stored procedure through the SQLDA must be in DBCS format.
- If an application calls a stored procedure through the Database Application Remote Interface (DARI) interface (the `sqleproc()` API), any graphic data in the input SQLDA must be in DBCS format, regardless of the state of the calling application's WCHARTYPE setting. Likewise, any graphic data in the output SQLDA will be returned in DBCS format, also regardless of the WCHARTYPE setting.
- If an application calls a stored procedure through the SQL CALL statement, graphic data conversion will occur on the SQLDA, depending on the calling application's WCHARTYPE setting.
- Graphic data passed to user-defined functions (UDFs) will always be in DBCS format. Likewise, any graphic data returned from a UDF will be assumed to be in DBCS format for DBCS databases, and UCS-2 format for EUC databases.
- Data stored in DBCLOB files, through the use of DBCLOB file reference variables, will always be in DBCS format. Likewise, input data retrieved from DBCLOB files will always be assumed to be in DBCS format.

Notes:

1. If you precompile C applications using the WCHARTYPE CONVERT option, DB2 validates the applications' graphic data on both input and output as the data is passed through the conversion functions. If you do **not** use the CONVERT option, no conversion of graphic data, and hence no validation occurs. In a mixed CONVERT/NOCONVERT environment, this may cause problems if invalid graphic data is inserted by a NOCONVERT application and then fetched by a CONVERT application. This data fails the conversion with an SQLCODE -1421 (SQLSTATE 22504) on a FETCH in the CONVERT application.
2. The WCHARTYPE CONVERT precompile option is not currently supported for programs running on the DB2 Windows 3.1 client. In this case, use the default WCHARTYPE NOCONVERT option.

Japanese or Traditional-Chinese EUC Considerations

If your application code page is Japanese or Traditional-Chinese EUC, you can access GRAPHIC columns at a database server by using either the CONVERT or the NOCONVERT option, and `wchar_t` or `sqldbchar` graphic host variables, or input/output SQLDAs. In this section, *DBCS format* refers to the UCS-2 encoding scheme for EUC data. Consider the following cases:

- CONVERT option used.

  The Client Application Enabler converts graphic data from the wide character format to EUC, then to UCS-2 before sending the input SQLDA to the database server. Any graphic data is sent to the database server tagged with the UCS-2 code page identifier. Mixed character data is tagged with the EUC code page identifier. When graphic data is retrieved from a database by a client running under the Japanese or Traditional-Chinese EUC code set, it is tagged with the UCS-2 code page identifier. The Client Application Enabler then converts the data from UCS-2 to EUC, then to the wide character format. If an input SQLDA is used instead of a host variable, then you are required to ensure that graphic data is encoded using the wide character format. This data will be converted to UCS-2 and then sent to the database server. These conversions will impact performance.

- NOCONVERT option used.

  The graphic data is assumed by DB2 to be encoded using UCS-2 and is tagged with the UCS-2 code page, and no conversions are done. DB2 assumes that the graphic host variable is being used simply as a bucket. When the NOCONVERT option is chosen, graphic data retrieved from the database server is passed to the application encoded using UCS-2. Any conversions from EUC to UCS-2 and from UCS-2 to EUC are your responsibility. Data tagged as UCS-2, is sent to the database server without any conversions or alterations.

To minimize conversions you can either use the NOCONVERT option and handle the conversions in your application, or not use GRAPHIC columns. Do not assign IBM-eucJP/IBM-eucTW CS0 (7-bit ASCII) and IBM-eucJP CS2 (Katakana) data to graphic host variables either after conversion to UCS-2 (if NOCONVERT is specified) or by conversion to the wide character format (if CONVERT is specified). This is because characters in both of these EUC code sets become single-byte when converted from UCS-2 to PC DBCS.

For general EUC application development guidelines, see “Japanese and Traditional-Chinese EUC Code Set Considerations” on page 377.

Supported SQL Data Types

Certain predefined C and C++ data types correspond to the database manager column types. Only these C/C++ data types can be declared as host variables.

Table 16 shows the C/C++ equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Table 16 (Page 1 of 4). SQL Data Types Mapped to C/C++ Declarations

SQL Column Type[1]	**C/C++ Data Type**	**SQL Column Type Description**
SMALLINT (500 or 501)	short short int	16-bit signed integer
INTEGER (496 or 497)	long long int	32-bit signed integer
REAL[2] (480 or 481)	float	Single-precision floating point
DOUBLE[3] (480 or 481)	double	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	No exact equivalent; use double	Packed decimal (Consider using the CHAR and DECIMAL functions to manipulate packed decimal fields as character data.)
CHAR(1) (452 or 453)	char	Single character
CHAR(*n*) (452 or 453)	No exact equivalent; use char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length character string

Table 16 (Page 2 of 4). SQL Data Types Mapped to C/C++ Declarations

SQL Column Type[1]	**C/C++ Data Type**	**SQL Column Type Description**

VARCHAR(*n*)
(448 or 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=4 000
		Non null-terminated varying character string with 2-byte string length indicator

	Alternately use char[*n+1*] where n is large enough to hold the data 1<=*n*<=4 000	null-terminated variable-length character string **Note:** Assigned an SQL type of 460/461.

LONG VARCHAR
(456 or 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	4 001<=
	n<=32 700
		Non null-terminated varying character string with 2-byte string length indicator

CLOB(*n*) (408 or 409)	sql type is clob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying character string with 4-byte string length indicator
CLOB locator variable[4] (964 or 965)	sql type is clob_locator	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (808 or 809)	sql type is clob_file	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	sql type is blob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying binary string with 4-byte string length indicator
BLOB locator variable[4] (960 or 961)	sql type is blob_locator	Identifies BLOB entities on the server
BLOB file reference variable[4] (804 or 805)	sql type is blob_file	Descriptor for the file containing BLOB data
DATE (384 or 385)	null-terminated character form	Allow at least 11 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 10 characters.
TIME (388 or 389)	null-terminated character form	Allow at least 9 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 8 characters.

Table 16 (Page 3 of 4). SQL Data Types Mapped to C/C++ Declarations

SQL Column Type[1]	**C/C++ Data Type**	**SQL Column Type Description**
TIMESTAMP (392 or 393)	null-terminated character form	Allow at least 27 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 26 characters.
Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.		
GRAPHIC(1) (468 or 469)	sqldbchar	Single double-byte character
GRAPHIC(*n*) (468 or 469)	No exact equivalent; use sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length double-byte character string

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	1<=*n*<=2 000
		Non null-terminated varying double-byte character string with 2-byte string length indicator

	Alternately use char[*n+1*] where n is large enough to hold the data 1<=*n*<=2 000	null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC
(472 or 473)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	2 001<= *n*<=16 350
		Non null-terminated varying double-byte character string with 2-byte string length indicator

Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE CONVERT option.		
GRAPHIC(1) (468 or 469)	wchar_t	• Single wide character (for C-type) • Single double-byte character (for column type)
GRAPHIC(*n*) (468 or 469)	No exact equivalent; use wchar_t [*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length double-byte character string

Table 16 (Page 4 of 4). SQL Data Types Mapped to C/C++ Declarations

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	1<=*n*<=2 000
		Non null-terminated varying double-byte character string with 2-byte string length indicator

	Alternately use char[*n+1*] where n is large enough to hold the data 1<=*n*<=2 000	null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC
(472 or 473)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	2 001<= *n*<=16 350
		Non null-terminated varying double-byte character string with 2-byte string length indicator

Note: The following data types are only available in the DBCS or EUC environment.		
DBCLOB(*n*) (412 or 413)	sql type is dbclob(*n*) 1<=*n*<=1 073 741 823	Non null-terminated varying double-byte character string with 4-byte string length indicator
DBCLOB locator variable[4] (968 or 969)	sql type is dbclob_locator	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (812 or 813)	sql type is dbclob_file	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where 0 < *n* < 25 is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where 24 < *n* < 54 is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following is a sample SQL declare section with host variables declared for supported SQL data types.

```
EXEC SQL BEGIN DECLARE SECTION;
...
     short     age = 26;               /* SQL type  500 */
     short     year;                   /* SQL type  500 */
     long      salary;                 /* SQL type  496 */
     long      deptno;                 /* SQL type  496 */
     float     bonus;                  /* SQL type  480 */
     double    wage;                   /* SQL type  480 */
     char      mi;                     /* SQL type  452 */
     char      name[6];                /* SQL type  460 */
     struct   {
               short len;
               char data[24];
              } address;               /* SQL type  448 */
     struct   {
               short len;
               char data[5764];
              } voice;                 /* SQL type  456 */
     sql type is clob(1m)
               chapter;                /* SQL type  408 */
     sql type is clob_locator
               chapter_locator;        /* SQL type  964 */
     sql type is clob_file
               chapter_file_ref;       /* SQL type  808 */
     sql type is blob(1m)
               video;                  /* SQL type  404 */
     sql type is blob_locator
               video_locator;          /* SQL type  960 */
     sql type is blob_file
               video_file_ref;         /* SQL type  804 */
     sql type is dbclob(1m)
               tokyo_phone_dir;        /* SQL type  412 */
     sql type is dbclob_locator
               tokyo_phone_dir_lctr;   /* SQL type  968 */
     sql type is dbclob_file
               tokyo_phone_dir_flref;  /* SQL type  812 */
     struct   {
               short len;
               sqldbchar data[100];
              } vargraphic1;           /* SQL type  464 */
                                       /* Precompiled with
                                       WCHARTYPE NOCONVERT option */
     struct   {
               short len;
               wchar_t data[100];
              } vargraphic2;           /* SQL type  464 */
                                       /* Precompiled with
                                       WCHARTYPE CONVERT option */
     struct   {
               short len;
               sqldbchar data[10000];
              } long_vargraphic1;      /* SQL type  472 */
                                       /* Precompiled with
                                       WCHARTYPE NOCONVERT option */
     struct   {
               short len;
               wchar_t data[10000];
              } long_vargraphic2;      /* SQL type  472 */
                                       /* Precompiled with
```

```
                                         WCHARTYPE CONVERT option */
       sqldbchar graphic1[100];          /* SQL type  468 */
                                         /* Precompiled with
                                         WCHARTYPE NOCONVERT option */
       wchar_t   graphic2[100];          /* SQL type  468 */
                                         /* Precompiled with
                                         WCHARTYPE CONVERT option */
       char      date[11];               /* SQL type  384 */
       char      time[9];                /* SQL type  388 */
       char      timestamp[27];          /* SQL type  392 */
       short     wage_ind;               /* Null indicator */
  ...
  EXEC SQL END DECLARE SECTION;
```

The following are additional rules for supported C/C++ data types:

- The data type `char` can be declared as `char` or `unsigned char`.
- The database manager processes null-terminated variable-length character string data type char[*n*] (data type 460), as VARCHAR(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in char[*n*] or the number of bytes preceding the first null-terminator (\0), whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of bytes preceding the first null-terminator (\0).
- The database manager processes null-terminated, variable-length graphic string data type, `wchar_t[`*n*`]` or `sqldbchar[`*n*`]` (data type 400), as VARGRAPHIC(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in `wchar_t[`*n*`]` or `sqldbchar[`*n*`]`, or the number of characters preceding the first graphic null-terminator, whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of characters preceding the first graphic null-terminator.
- Unsigned numeric data types are not supported.
- The C/C++ data type `int` is not allowed since its internal representation is machine dependent.

FOR BIT DATA

The standard C or C++ string type 460 should not be used for columns designated FOR BIT DATA. The database manager truncates this data type when a null character is encountered. Use either the VARCHAR (SQL type 448) or CLOB (SQL type 408) structures.

SQLSTATE and SQLCODE Variables

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables :

```
EXEC SQL BEGIN DECLARE SECTION;
  char  SQLSTATE[6]
  long  SQLCODE;
  .
  .
  .
EXEC SQL END DECLARE SECTION;
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

In an application that is made up of multiple source files, the SQLCODE and SQLSTATE variables may be defined in the first source file as above. Subsequent source files should modify the definitions as follows:

```
extern long SQLCODE;
extern char SQLSTATE[6];
```

Chapter 12. Programming in COBOL

This chapter discusses special programming considerations for database manager applications written in COBOL. This includes the following topics:

- Language Restrictions
- Input and Output Files
- Include Files
- Embedding SQL Statements
- Host Variables
- Supported SQL Data Types.

Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on language restrictions, host language specific include files, embedding SQL statements, host variables, and supported data types for host variables.

Language Restrictions

All API pointers are 4 bytes long. All integer variables used as value parameters in API calls must be declared with a USAGE COMP-5 clause.

Input and Output Files

By default, the input file has an extension of `.sqb`, but if you use the TARGET precompile option (TARGET ANSI_COBOL, TARGET IBMCOB, TARGET MFCOB or TARGET MFCOB16), the input file can have any extension you prefer.

By default, the output file has an extension of `.cbl`, but you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

Include Files

The host-language-specific include files for COBOL have the file extension `.cbl`. The include files that are intended to be used in your applications are described below.

SQL (sql.cbl) This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLAPREP (sqlaprep.cbl) This file contains definitions required to write your own precompiler.

SQLCA (sqlca.cbl) This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

SQLCA_92 (sqlca_92.cbl)
This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of the `sqlca.cbl` file when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.cbl` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

SQLCODES (sqlcodes.cbl)
This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqlda.cbl)
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

SQLEAU (sqleau.cbl)
This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

SQLENV (sqlenv.cbl)
This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLETSD (sqletsd.cbl)
This file defines the Table Space Descriptor structure, SQLETSDESC, which is passed to the Create Database API, sqlgcrea.

SQLE819A (sqle819a.cbl)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.cbl)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.cbl)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.cbl)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC

US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.cbl) If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.cbl) If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQL1252A (sql1252a.cbl) If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQL1252B (sql1252b.cbl) If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLMON (sqlmon.cbl) This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLMONCT (sqlmonct.cbl) This file contains constant definitions and local data structure definitions required to call the Database System Monitor APIs.

SQLSTATE (sqlstate.cbl) This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLUTBCQ (sqlutbcq.cbl) This file defines the Table Space Container Query data structure, SQLB-TBSCONTQRY-DATA, which is used with the table space container query APIs, sqlgstsc, sqlgftcq and sqlgtcq.

SQLUTBSQ (sqlutbsq.cbl) This file defines the Table Space Query data structure, SQLB-TBSQRY-DATA, which is used with the table space query APIs, sqlgstsq, sqlgftsq and sqlgtsq.

SQLUTIL (sqlutil.cbl) This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

Embedding SQL Statements

Embedded SQL statements consist of the following three elements:

Element	Correct COBOL Syntax
Keyword pair	EXEC SQL
Statement string	Any valid SQL statement
Statement terminator	END-EXEC.

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

The following rules apply to embedded SQL statements:

- Executable SQL statements must be placed in the PROCEDURE DIVISION. The SQL statements can be preceded by a paragraph name just as a COBOL statement.
- SQL statements can begin in either Area A (columns 8 through 11) or Area B (columns 12 through 72).
- Start each SQL statement with EXEC SQL and end it with END-EXEC. The SQL precompiler includes each SQL statement as a comment in the modified source file.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end. Do not place SQL comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the COBOL language.
- COBOL comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
- SQL statements follow the same line continuation rules as the COBOL language. However, do not split the EXEC SQL keyword pair between lines.
- Do not use the COBOL COPY statement to include files containing SQL statements. SQL statements are precompiled before the module is compiled. The precompiler will ignore the COBOL COPY statement. Instead, use the SQL INCLUDE statement to include these files.

  To locate the INCLUDE file, the DB2 COBOL precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

  - `EXEC SQL INCLUDE payroll END-EXEC.`

If the file specified in the INCLUDE statement is not enclosed in quotes, as above, the precompiler searches for `payroll.sqb`, then `payroll.cpy`, then `payroll.cbl`, in each directory in which it looks.

- `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

  If the file name is enclosed in quotes, as above, no extension is added to the name.

  If the file name in quotes does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for AIX, if DB2INCLUDE is set to '`/disk2:myfiles/cobol`', the precompiler searches for '`./pay/payroll.cbl`', then '`/disk2/pay/payroll.cbl`', and finally '`./myfiles/cobol/pay/payroll.cbl`'. The path where the file is actually found is displayed in the precompiler messages. On OS/2 and Windows platforms, substitute back slashes (\) for the forward slashes in the above example.

Note: The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

- To continue a string constant to the next line, column 7 of the continuing line must contain a `'-'` and column 12 or beyond must contain a string delimiter.
- SQL arithmetic operators must be delimited by blanks.
- Full-line COBOL comments can occur anywhere in the program, including within SQL statements.
- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters such as end-of-line and TAB characters occur as follows:
  - When they occur outside quotes (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotes, the end-of-line characters disappear, provided the string is continued properly for a COBOL program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

Host Variables

Host variables are COBOL language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other COBOL variable. Obey the rules described below when naming, declaring, and using host variables.

Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with a prefix other than SQL, which is reserved for system use.
- FILLER items using the declaration syntaxes described below are permitted in group host variable declarations, and will be ignored by the precompiler. You may not include FILLER items in VARCHAR, LONG VARCHAR, VARGRAPHIC or LONG VARGRAPHIC declarations.
- You can use hyphens in host variable names.

  SQL interprets a hyphen enclosed by spaces as a subtraction operator. Use hyphens without spaces in host variable names.
- The REDEFINES clause is permitted in host variable declarations.
- Level-88 declarations are permitted in the host variable declare section, but are ignored.

Declaring Host Variables

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The COBOL precompiler only recognizes a subset of valid COBOL declarations.

Figure 37 shows the syntax for numeric host variables.

```
                                           ┌─IS─┐
►►──┬─01─┬──variable-name──┬─PICTURE─┬──┴────┴──picture-string──────────►
    └─77─┘                 └─PIC─────┘

►──┬────────────────────────────────────────────────┬────────────────────►
   └──┬─────────────────┬──┬─COMP-3─(1)──────────┬──┘
      │       ┌─IS─┐    │  ├─COMPUTATIONAL-3─────┤
      └─USAGE─┴────┴────┘  ├─COMP-5──────────────┤
                           └─COMPUTATIONAL-5─────┘

►──┬──────────────────────────┬──.──►◄
   │       ┌─IS─┐             │
   └─VALUE─┴────┴──value──────┘
```

Note:

[1] An alternative for COMP-3 is PACKED-DECIMAL.

Floating Point

```
►►──┬─01─┬──variable-name──┬────────────────────┬──────────────────────►
    └─77─┘                 │        ┌─IS─┐      │
                           └─USAGE──┴────┴──────┘

►──┬─┬─COMPUTATIONAL-1─┬─(1)─┬──┬──────────────────────────┬──.──────────►◄
   │ └─COMP-1──────────┘     │  │       ┌─IS─┐             │
   └─┬─COMPUTATIONAL-2─┬─(2)─┘  └─VALUE─┴────┴──value──────┘
     └─COMP-2──────────┘
```

Notes:

[1] REAL (SQLTYPE 480), Length 4

[2] DOUBLE (SQLTYPE 480), Length 8

Figure 37. Syntax for Numeric Host Variables in COBOL

Notes:

1. *Picture-string* must have one of the following forms:
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. Nines may be expanded (e.g., "S999" instead of S9(3)")
3. *m* and *n* must be positive integers.

Figure 38 shows the syntax for character host variables.

```
Fixed Length

                                               ┌─IS─┐
►►──┬─01─┬──variable-name──┬─PICTURE─┬─────────┴────┴──picture-string──────────►
    └─77─┘                 └─PIC─────┘

►──┬─────────────────────────────┬──.──►◄
   │          ┌─IS─┐             │
   └─VALUE────┴────┴──value──────┘

Variable Length

►►──01──variable-name──.──►◄


                                           ┌─IS─┐
►►──49──identifier-1──┬─PICTURE─┬──────────┴────┴──S9(4)──────────────────────►
                      └─PIC─────┘

►──┬──────────────────────────────────────────────┬───────────────────────────►
   └─┬──────────────────┬──┬─COMP-5─────────────┬──┘
     │       ┌─IS─┐     │  └─COMPUTATIONAL-5─┘
     └─USAGE─┴────┴─────┘

►──┬─────────────────────────────┬──.──►◄
   │          ┌─IS─┐             │
   └─VALUE────┴────┴──value──────┘


                                           ┌─IS─┐
►►──49──identifier-2──┬─PICTURE─┬──────────┴────┴──picture-string─────────────►
                      └─PIC─────┘

►──┬─────────────────────────────┬──.──►◄
   │          ┌─IS─┐             │
   └─VALUE────┴────┴──value──────┘
```

Figure 38. Syntax for Character Host Variables in COBOL

Notes:

1. *Picture-string* must have the form *X(m)*. Alternately, X's may be expanded (for example, "XXX" instead of "X(3)").
2. *m* is from 1 to 254 for fixed-length strings.
3. *m* is from 1 to 32 700 for variable-length strings.
4. If *m* is greater than 4 000, the host variable will be treated as a LONG VARCHAR string, and its use may be restricted.
5. Use X and 9 as the picture characters in any PICTURE clause. Other characters are not allowed.
6. Variable-length strings consist of a length item and a value item. You can use acceptable COBOL names for the length item and the string item. However, refer to the variable-length string by the collective name in SQL statements.

7. In a CONNECT statement, such as shown below, COBOL character string host variables dbname and userid will have any trailing blanks removed before processing:

   ```
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

   However, because blanks can be significant in passwords, the p-word host variable should be declared as a VARCHAR data item, so that your application can explicitly indicate the significant password length for the CONNECT statement as follows:

   ```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 dbname PIC X(8).
   01 userid PIC X(8).
   01 p-word.
       49 L PIC S9(4) COMP-5.
       49 D PIC X(18).
   EXEC SQL END DECLARE SECTION END-EXEC.
   PROCEDURE DIVISION.
       MOVE "sample" TO dbname.
       MOVE "userid" TO userid.
       MOVE "password" TO D OF p-word.
       MOVE 8          TO L of p-word.
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

Figure 39 shows the syntax for graphic host variables.

```
Fixed Length

                                                 ┌─IS─┐
►►──┬─01─┬──variable-name──┬─PICTURE─┬───┴────┴──picture-string──USAGE────►
    └─77─┘                 └─PIC─────┘

    ┌─IS─┐
►───┴────┴──DISPLAY-1──┬────────────────────────────┬──.──►◄
                       │              ┌─IS─┐        │
                       └─VALUE────────┴────┴──value─┘

Variable Length

►►──01──variable-name──.──►◄


                                        ┌─IS─┐
►►──49──identifier-1──┬─PICTURE─┬───────┴────┴──S9(4)─────────────────────►
                      └─PIC─────┘

►───┬───────────────────────────────────────────────────┬─────────────────►
    │                 ┌─IS─┐                            │
    └──┬──────────────┴────┴───┬──┬─COMP-5────────────┬─┘
       └─USAGE─────────────────┘  └─COMPUTATIONAL-5───┘

►───┬──────────────────────────────┬──.──►◄
    │              ┌─IS─┐          │
    └─VALUE────────┴────┴──value───┘


                                        ┌─IS─┐
►►──49──identifier-2──┬─PICTURE─┬───────┴────┴──picture-string──USAGE─────►
                      └─PIC─────┘

    ┌─IS─┐
►───┴────┴──DISPLAY-1──┬────────────────────────────┬──.──►◄
                       │              ┌─IS─┐        │
                       └─VALUE────────┴────┴──value─┘
```

Figure 39. Syntax for Graphic Host Variables in COBOL

Notes:

1. *Picture-string* must have the form *G(m)*. Alternately, G's may be expanded (for example, "GGG" instead of "G(3)").
2. *m* is from 1 to 127 for fixed-length strings.
3. *m* is from 1 to 16 350 for variable-length strings.
4. If *m* is greater than 2 000, the host variable will be treated as a LONG VARGRAPHIC string, and its use may be restricted.

Indicator Variables

Indicator variables should be declared as a `PIC S9(4) COMP-5` data type.

LOB Declarations in COBOL

Figure 40 shows the syntax for declaring large object (LOB) host variables in COBOL.

```
►►──01──variable-name──┬──────────────────┬──SQL TYPE IS──┬─BLOB───┬──────────►
                       └─USAGE──┬──────┬──┘               ├─CLOB───┤
                                └─IS───┘                  └─DBCLOB─┘

►──(──length──┬───┬──)──.──────────────────────────────────────────────────►◄
              ├─K─┤
              ├─M─┤
              └─G─┘
```

Figure 40. Syntax for LOB Host Variables in COBOL

Notes:

1. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
2. For DBCLOB 1 <= lob-length <= 1 073 741 823.
3. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in either uppercase, lowercase, or mixed.
4. Initialization within the LOB declaration is not permitted.
5. The host variable name prefixes LENGTH and DATA in the precompiler generated code.

BLOB Example:

Declaring:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

CLOB Example:

Declaring:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

DBCLOB Example:

Declaring:

```
        01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

Results in the generation of the following structure:

```
   01 MY-DBCLOB.
      49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
      49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

LOB Locator Declarations in COBOL

Figure 41 shows the syntax for declaring large object (LOB) locator host variables in COBOL.

```
►►──01──variable-name──┬─────────────────┬──SQL TYPE IS──────────────────────►
                       └─USAGE──┬────┬───┘
                                └─IS─┘

►──┬─BLOB-LOCATOR───┬──.──────────────────────────────────────────────────►◄
   ├─CLOB-LOCATOR───┤
   └─DBCLOB-LOCATOR─┘
```

Figure 41. Syntax for LOB Locator Host Variables in COBOL

Notes:

1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR can be either uppercase, lowercase, or mixed.
2. Initialization of locators is not permitted.

BLOB Locator Example (other LOB locator types are similar):

Declaring:

```
   01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

Results in the generation of the following declaration:

```
   01 MY-LOCATOR PIC S9(9) COMP-5.
```

File Reference Declarations in COBOL

Figure 42 shows the syntax for declaring file reference host variables in COBOL.

```
►►──01──variable-name──┬──────────────────┬──SQL TYPE IS──┬─BLOB-FILE───┬──────►
                       └─USAGE──┬──────┬──┘               ├─CLOB-FILE───┤
                                └─IS───┘                  └─DBCLOB-FILE─┘
►──.──────────────────────────────────────────────────────────────────────────►◄
```

Figure 42. Syntax for File Reference Host Variables in COBOL

- SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be either uppercase, lowercase, or mixed.

BLOB File Reference Example (other LOB types are similar):

Declaring:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

Results in the generation of the following declaration:

```
01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

Host structure support in COBOL

The COBOL precompiler supports declarations of group data items in the host variable declare section. Among other things, this provides a shorthand for referring to a set of elementary data items in an SQL statement. For example, the following group data item can be used to access the STAFF table in the SAMPLE database:

```
01 staff-record.
    05 staff-id      pic s9(4) comp-5.
    05 staff-name.
        49 l         pic s9(4) comp-5.
        49 d         pic x(9).
    05 staff-dept    pic s9(4) comp-5.
    05 staff-job     pic x(5).
    05 staff-years   pic s9(4) comp-5.
    05 staff-salary  pic s9(5)v99 comp-3.
    05 staff-comm    pic s9(5)v99 comp-3.
```

Group data items in the declare section can have any of the valid host variable types described above as subordinate data items. This includes all numeric and character types, as well as all large object types. Group data items cannot be nested. The exception is the `varchar` character type, shown in `staff-name` above. It is permitted within a group data item because the precompiler treats it as a single host variable instead of a group. Note that you must declare `varchar` character types with the subordinate items at level 49, as in the above example. If they are not at level 49, the `varchar` is treated as a group data item with two subordinates, and is subject to the rules of declaring and using group data items. You must declare group data items at the 01-level, and the subordinate items within them at a common level between 02 and 49. For example, all the subordinates in staff-record above are at the 05 level. FILLER items are not permitted within group data items.

You can use group data items and their subordinates in three ways:

- Method 1.

  The entire group may be referenced as a single host variable in an SQL statement:

  ```
  EXEC SQL SELECT ID,NAME,DEPT,JOB,YEARS,SALARY,COM
    INTO :staff-record
    FROM STAFF WHERE ID = 10 END-EXEC.
  ```

  The precompiler converts the reference to staff-record into a list, separated by commas, of all the subordinate items declared within staff-record. Each elementary item is qualified with the group name to prevent naming conflicts with other items. This is equivalent to the following method.

- Method 2.

  The second way of using group data items:

  ```
  EXEC SQL SELECT id,name,dept,job,years,salary,com
    INTO
    :staff-record.staff-id,
    :staff-record.staff-name,
    :staff-record.staff-dept,
    :staff-record.staff-job,
    :staff-record.staff-years,
    :staff-record.staff-salary,
    :staff-record.staff-comm
    FROM staff WHERE id = 10 END-EXEC.
  ```

  Note that the reference to staff-id is qualified with its group name, using the staff-record. prefix, and not staff-id of staff-record as in pure COBOL. Assuming there are no other host variables with the same names as the subordinates of staff-record, the above statement can also be coded as in method 3, eliminating the explicit group qualification.

- Method 3.

  Here, subordinate items are referenced in a typical COBOL fashion, without being qualified to their particular group item.

  ```
  EXEC SQL SELECT ID, NAME, DEPT, JOB, YEARS, SALARY, COM
    INTO
    :staff-id,
    :staff-name,
    :staff-dept,
    :staff-job,
    :staff-years,
    :staff-salary,
    :staff-comm
    FROM STAFF WHERE ID = 10 END-EXEC.
  ```

As in pure COBOL, this method is acceptable to the precompiler as long as a given subordinate item only occurs in one group declared as a host variable. If, for example, staff-id occurs in more than one group, the precompiler issues an error indicating an ambiguous reference:

```
SQL0088N Host variable "staff-id" is ambiguous.
```

Because a reference to a group item alone, as in method 1, is equivalent to a comma-separated list of its subordinates, there are instances where this type of reference leads to an error. For example:

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

Here, the CONNECT statement expects a single character-based host variable. By giving the staff-record group data item instead, the host variable results in the following precompile-time error:

```
SQL0087N Host variable "staff-record" is a structure used where
         structure references are not permitted.
```

Other uses of group items which cause an SQL0087N to occur include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables, and SQLDA references. Groups with only one subordinate are permitted in such situations, as are references to individual subordinates, as in method 2 and 3 above.

Indicator Tables

The COBOL precompiler supports the declaration of tables of indicator variables, which are convenient to use with group data items. They are declared as follows:

```
01 <indicator-table-name>.
   05 <indicator-name> pic s9(4) comp-5
                       occurs <table-size> times.
```

For example:

```
01 staff-indicator-table.
   05 staff-indicator pic s9(4) comp-5
                      occurs 7 times.
```

This indicator table can be used effectively with the first format of group item reference above:

```
EXEC SQL SELECT ID, NAME, DEPT, JOB, YEARS, SALARY, COM
  INTO :staff-record :staff-indicator
  FROM STAFF WHERE ID = 10 END-EXEC.
```

Here, the precompiler detects that staff-indicator was declared as an indicator table, and expands it into individual indicator references when it processes the SQL statement. staff-indicator(1) is associated with staff-id of staff-record, staff-indicator(2) is associated with staff-name of staff-record, and so on.

Note: If there are k more indicator entries in the indicator table than there are subordinates in the data item, (for example, if staff-indicator has 10 entries, making k=3), the k extra entries at the end of the indicator table are ignored. Likewise, if there are k fewer indicator entries than subordinates, the last k subordinates in the group item do not have indicators associated with them. Note that you cannot individually refer to elements in an indicator table in an SQL statement.

Using BINARY/COMP-4 COBOL Data Types

The DB2 COBOL precompiler supports the use of BINARY, COMP, and COMP-4 data types wherever integer host variables and indicators are permitted, as long as the target COBOL compiler views (or can be made to view) the BINARY, COMP, or COMP-4 data types as equivalent to the COMP-5 data type. In this book, such host variables and indicators are shown with type COMP-5. Target compilers supported by DB2 that treat COMP, COMP-4, BINARY COMP and COMP-5 as equivalent are:

- IBM COBOL Set for AIX
- Micro Focus COBOL for AIX
- IBM COBOL Visual Set for OS/2 (with the -qbinary(native) option set)
- IBM VisualAge for COBOL for OS/2, Windows NT and Windows 95, (with the -qbinary(native) option set)

Using REDEFINES in COBOL Group Data Items

You can use the REDEFINES clause when declaring host variables. If you declare a member of a group data item with the REDEFINES clause and that group data item is referred to as a whole in an SQL statement, any subordinate items containing the REDEFINES clause are not expanded. For example:

```
01 foo.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

Referring to foo in an SQL statement as follows:

```
... INTO :foo ...
```

The above statement is equivalent to:

```
... INTO :foo.a, :foo.b ...
```

That is, the subordinate item a1, declared with the REDEFINES clause is not automatically expanded out in such situations. If a1 is unambiguous, you can explicitly refer to a subordinate with a REDEFINES clause in an SQL statement, as follows:

```
... INTO :foo.a1 ...
```

or

```
... INTO :a1 ...
```

Supported SQL Data Types

Certain predefined COBOL data types correspond to column types. Only these COBOL data types can be declared as host variables.

Table 17 on page 469 shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

Table 17 (Page 1 of 2). SQL Data Types Mapped to COBOL Declarations

SQL Column Type[1]	**COBOL Data Type**	**SQL Column Type Description**
SMALLINT (500 or 501)	01 name PIC S9(4) COMP-5.	16-bit signed integer
INTEGER (496 or 497)	01 name PIC S9(9) COMP-5.	32-bit signed integer
DECIMAL(*p*,*s*) (484 or 485)	01 name PIC S9(*m*) V9(*n*) COMP-3.	Packed decimal
REAL[2] (480 or 481)	01 name USAGE IS COMP-1.	Single-precision floating point
DOUBLE[3] (480 or 481)	01 name USAGE IS COMP-2.	Double-precision floating point
CHAR(*n*) (452 or 453)	01 name PIC X(*n*).	Fixed-length character string
VARCHAR(*n*) (448 or 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=4000	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 4001<=*n*<=32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (808 or 809)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	Identifies BLOB entities residing on the server

Table 17 (Page 2 of 2). SQL Data Types Mapped to COBOL Declarations

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
BLOB file reference variable[4] (804 or 805)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
DATE (384 or 385)	01 identifier PIC X(10).	10-byte character string
TIME (388 or 389)	01 identifier PIC X(8).	8-byte character string
TIMESTAMP (392 or 393)	01 identifier PIC X(26).	26-byte character string
Note: The following data types are only available in the DBCS environment.		
GRAPHIC(*n*) (468 or 469)	01 name PIC G(*n*) DISPLAY-1.	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=2 000	Variable length double-byte character string with 2-byte string length indicator
LONG VARGRAPHIC (472 or 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 2001<=*n*<=16 350	Variable length double-byte character string with 2-byte string length indicator
DBCLOB(*n*) (412 or 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n). 1<=*n*<=1 073 741 823	Large object variable length double-byte character string with 4-byte string length indicator
DBCLOB locator variable[4] (968 or 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (812 or 813)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where 0 < *n* < 25 is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where 24 < *n* < 54 is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following is a sample SQL declare section with a host variable declared for each supported SQL data type.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
  01 age        PIC S9(4) COMP-5.
  01 divis      PIC S9(9) COMP-5.
  01 salary     PIC S9(6)V9(3) COMP-3.
  01 bonus      USAGE IS COMP-1.
  01 wage       USAGE IS COMP-2.
  01 nm         PIC X(5).
  01 varchar.
     49 leng    PIC S9(4) COMP-5.
     49 strg    PIC X(14).
  01 longvchar.
     49 len     PIC S9(4) COMP-5.
     49 str     PIC X(6027).
  01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).
  01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.
  01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.
  01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).
  01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.
  01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.
  01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).
  01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.
  01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.
  01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.
  01 dt         PIC X(10).
  01 tm         PIC X(8).
  01 tmstmp     PIC X(26).
  01 wage-ind   PIC S9(4) COMP-5.
*
EXEC SQL END DECLARE SECTION END-EXEC.
```

The following are additional rules for supported COBOL data types:

- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. Refer to the following sample:

  ```
  01 identifier PIC S9(m)V9(n) COMP-3
  ```

  – Use V to denote the decimal point.
  – Values for *n* and *m* must be greater than or equal to 1.
  – The value for *n* + *m* cannot exceed 31.
  – The value for *s* equals the value for *n*.
  – The value for *p* equals the value for *n* + *m*.
  – The repetition factors *(n)* and *(m)* are optional. The following examples are all valid:

```
01 identifier PIC S9(3)V COMP-3
01 identifier PIC SV9(3) COMP-3
01 identifier PIC S9V COMP-3
01 identifier PIC SV9 COMP-3
```

  – PACKED-DECIMAL can be used instead of COMP-3.

- The following are not supported by the COBOL precompiler:
  – The database manager column type FLOAT (SQL type 480)
  – Arrays

FOR BIT DATA

Certain database columns can be declared FOR BIT DATA. These columns, which generally contain characters, are used to hold binary information. The CHAR(*n*), VARCHAR, LONG VARCHAR, and BLOB data types are the COBOL host variable types that can contain binary data. Use these data types when working with columns with the FOR BIT DATA attribute.

SQLSTATE and SQLCODE Variables

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables :

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PICTURE X(5).
01 SQLCODE  PICTURE S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. The '01' can also be '77' and the 'PICTURE' can be 'PIC'. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications made up of multiple source files, the SQLCODE and SQLSTATE declarations may be included in each source file as shown above.

Japanese or Traditional-Chinese EUC Considerations

Any graphic data sent from your application running under an EUC code set is tagged with the UCS-2 code page identifier, so your application must convert an EUC string to UCS-2 before sending it to a database server. Likewise, graphic data retrieved from a database by your application running under an EUC code is encoded using UCS-2. This requires your application to convert from UCS-2 to EUC internally, unless the user is to be presented with UCS-2 data. You are responsible for converting to and from UCS-2 since this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. No application-accessible conversion routines are supplied with the DB2 Universal Database products. Use system calls available from your operating system.

For general EUC application development guidelines, see “Japanese and Traditional-Chinese EUC Code Set Considerations” on page 377.

Object Oriented COBOL

If you are using Object Oriented COBOL, you must observe the following:

- SQL statements can only appear in the first program or class in a compile unit. This is because the precompiler inserts temporary working data into the first Working-Storage section it sees.
- In an Object Oriented COBOL program, every class containing SQL statements must have a class-level Working-Storage Section, even if it is empty. This section is used to store data definitions generated by the precompiler.

Chapter 13. Programming in FORTRAN

This chapter discusses programming considerations for database manager applications written in FORTRAN. This includes the following topics:

- Language Restrictions
- Input and Output Files
- Include Files
- Embedding SQL Statements
- Host Variables
- Supported SQL Data Types
- Considerations for Multi-byte Character Sets
- Japanese or Traditional-Chinese EUC Considerations

Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on language restrictions, host language specific include files, embedding SQL statements, host variables, and supported data types for host variables.

Language Restrictions

The following sections describe the FORTRAN language restrictions.

Call by Reference

Some API parameters require addresses rather than values in the call variables. The database manager provides the GET ADDRESS, DEREFERENCE ADDRESS, and COPY MEMORY APIs which simplify your ability to provide these parameters. See the *API Reference* for a full description of these APIs.

Debugging and Comment Lines

Some FORTRAN compilers treat lines with a `'D'` or `'d'` in column 1 as conditional lines. These lines can either be compiled for debugging or treated as comments. The precompiler will always treat lines with a `'D'` or `'d'` in column 1 as comments.

Including Files

There are two methods for including files: the EXEC SQL INCLUDE statement and the FORTRAN INCLUDE statement. The precompiler will ignore FORTRAN INCLUDE statements, and only process files included with the EXEC SQL statement.

To locate the INCLUDE file, the DB2 FORTRAN precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll`

  If the file specified in the INCLUDE statement is not enclosed in quotes, as above, the precompiler searches for `payroll.sqf`, then `payroll.f` (`payroll.for` on OS/2) in each directory in which it looks.

- `EXEC SQL INCLUDE 'pay/payroll.f'`

  If the file name is enclosed in quotes, as above, no extension is added to the name. (For OS/2, the file would be specified as `'pay\payroll.for'`.)

  If the file name in quotes does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for AIX, if DB2INCLUDE is set to '`/disk2:myfiles/fortran`', the precompiler searches for '`./pay/payroll.f`', then '`/disk2/pay/payroll.f`', and finally '`./myfiles/cobol/pay/payroll.f`'. The path where the file is actually found is displayed in the precompiler messages. On OS/2, substitute back slashes (\) for the forward slashes, and substitute `'for'` for the `'f'` extension in the above example.

Note: The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

Precompiling Considerations

The following items affect the precompiling process:

- The precompiler allows only digits, blanks, and tab characters within columns 1-5 on continuation lines.
- Hollerith constants are not supported in `.sqf` source files.

See the *DB2 SDK Building Applications* book for your operating environment for any other precompiling considerations that may affect you.

Input and Output Files

By default, the input file has an extension of `.sqf`, but if you use the TARGET precompile option the input file can have any extension you prefer.

By default, the output file has an extension of `.f` on UNIX-platforms, and `.for` on OS/2 and Windows-based platforms, however you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

Include Files

The host-language specific include files for FORTRAN have the file extension `.f` on AIX and UNIX-based platforms, and `.for` on OS/2. For other platforms, consult the *DB2 SDK Building Applications* book for that platform for information on file extensions. The following are FORTRAN include files that are intended to be used in your applications.

SQL (sql.f)
This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLAPREP (sqlaprep.f)
This file contains definitions required to write your own precompiler.

SQLCA (sqlca_cn.f, sqlca_cs.f)
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

Two SQLCA files are provided for FORTRAN applications. The default, `sqlca_cs.f`, defines the SQLCA structure in an IBM SQL compatible format. The `sqlca_cn.f` file, precompiled with the `SQLCA NONE` option, defines the SQLCA structure for better performance.

SQLCA_92 (sqlca_92.f)
This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of either the `sqlca_cn.f` or the `sqlca_cs.f` files when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.f` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

SQLCODES (sqlcodes.f)
This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqldact.f)
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager. See "Allocating an SQLDA Structure" on page 107 for details of how to code an SQLDA in a FORTRAN program.

SQLEAU (sqleau.f)
This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

SQLENV (sqlenv.f)
This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLE819A (sqle819a.f)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.f)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.f)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.f)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.f)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.f)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQL1252A (sql1252a.f)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQL1252B (sql1252b.f)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary

collation. This file is used by the CREATE DATABASE API.

SQLMON (sqlmon.f)
This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLSTATE (sqlstate.f)
This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLUTIL (sqlutil.f)
This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

Embedding SQL Statements

Embedded SQL statements consist of the following three elements:

Element	**Correct FORTRAN Syntax**
Keyword	EXEC SQL
Statement string	Any valid SQL statement with blanks as delimiters
Statement terminator	End of source line.

The end of the source line serves as the statement terminator. If the line is continued, the statement terminator is the end of the last continued line.

For example:

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

The following rules apply to embedded SQL statements:

- Code SQL statements between columns 7 and 72 only.
- Use full-line FORTRAN comments, or SQL comments, but do not use the FORTRAN end-of-line comment '!' character in SQL statements. This comment character may be used elsewhere, including host variable declarations.
- Use blanks as delimiters when coding embedded SQL statements, even though FORTRAN statements do not require blanks as delimiters.
- Use only one SQL statement for each FORTRAN source line. Normal FORTRAN continuation rules apply for statements that require more than one source line. Do not split the EXEC SQL keyword pair between lines.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end.
- FORTRAN comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.

  - The extension of using ! to code a FORTRAN comment at the end of a line is not supported within an embedded SQL statement.
- Use exponential notation when specifying a real constant in SQL statements. The database manager interprets a string of digits with a decimal point in an SQL statement as a decimal constant, not a real constant.
- Statement numbers are invalid on SQL statements that precede the first executable FORTRAN statement. If an SQL statement has a statement number associated with it, the precompiler generates a labeled CONTINUE statement that directly precedes the SQL statement.
- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters such as end-of-line and TAB characters occur as follows:
  - When they occur outside quotes (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotes, the end-of-line characters disappear, provided the string is continued properly for a FORTRAN program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

Host Variables

Host variables are FORTRAN language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from it. After the application is precompiled, host variables are used by the compiler as any other FORTRAN variable. Use the following suggestions when naming, declaring, and using host variables.

Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following suggestions apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than EXEC or SQL. These prefixes are reserved for system use.

Declaring Host Variables

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The FORTRAN precompiler only recognizes a subset of valid FORTRAN declarations as valid host variable declarations. These declarations define either numeric or character variables. A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The programmer must ensure that output variables are long enough to contain the values that they will receive. Figure 43 shows the syntax for numeric host variables.

```
                           ┌─,─────────────────────────────────┐
►►──┬─INTEGER*2────────┬───▼─varname─┬──────────────────────┬──┴──────────────►◄
    ├─INTEGER*4────────┤             └─ / initial-value / ─┘
    ├─REAL*4───────────┤
    ├─REAL *8──────────┤
    └─DOUBLE PRECISION─┘
```

Figure 43. Syntax for Numeric Host Variables in FORTRAN

Notes:

1. REAL*8 and DOUBLE PRECISION are equivalent.
2. Use an E rather than a D as the exponent indicator for REAL*8 constants.

Figure 44 shows the syntax for character host variables.

```
Fixed Length

                           ┌─,─────────────────────────────────┐
►►──CHARACTER──┬────┬──────▼─varname─┬──────────────────────┬──┴──────────────►◄
               └─*n─┘                └─ / initial-value / ─┘

Variable Length

                                        ┌─,─────────┐
►►──SQL TYPE IS VARCHAR──(length)───────▼─varname───┴──────────────────────────►◄
```

Figure 44. Syntax for Character Host Variables in FORTRAN

Notes:

1. *n has a maximum value of 254.
2. When length is between 1 and 4000 inclusive, the host variable has type VARCHAR(448).
3. When length is between 4001 and 32700 inclusive, the host variable has type LONG VARCHAR(456).
4. Initialization of VARCHAR and LONG VARCHAR host variables is not permitted within the declaration.

VARCHAR Example:

Declaring:

```
sql type is varchar(1000) my_varchar
```

Results in the generation of the following structure:

```
 character    my_varchar(1000+2)
 integer*2    my_varchar_length
 character    my_varchar_data(1000)
 equivalence( my_varchar(1),
+             my_varchar_length )
 equivalence( my_varchar(3),
+             my_varchar_data )
```

The application may manipulate both my_varchar_length and my_varchar_data; for example, to set or examine the contents of the host variable. The base name (in this case, my_varchar), is used in SQL statements to refer to the VARCHAR as a whole.

LONG VARCHAR Example:

Declaring:

```
sql type is varchar(10000) my_lvarchar
```

Results in the generation of the following structure:

```
 character    my_lvarchar(10000+2)
 integer*2    my_lvarchar_length
 character    my_lvarchar_data(10000)
 equivalence( my_lvarchar(1),
+             my_lvarchar_length )
 equivalence( my_lvarchar(3),
+             my_lvarchar_data )
```

The application may manipulate both my_lvarchar_length and my_lvarchar_data; for example, to set or examine the contents of the host variable. The base name (in this case, my_lvarchar), is used in SQL statements to refer to the LONG VARCHAR as a whole.

Note: In a CONNECT statement, such as in the following example, FORTRAN character string host variables dbname and userid will have any trailing blanks removed before processing.

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

However, because blanks can be significant in passwords, you should declare host variables for passwords as VARCHAR, and have the length field set to reflect the actual password length:

```
EXEC SQL BEGIN DECLARE SECTION
  character*8 dbname, userid
  sql type is varchar(18) passwd
EXEC SQL END DECLARE SECTION
character*18 passwd_string
equivalence(passwd_data,passwd_string)
dbname = 'sample'
userid = 'userid'
passwd_length= 8
passwd_string = 'password'
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

Indicator Variables

Indicator variables should be declared as an `INTEGER*2` data type.

LOB Declarations in FORTRAN

Figure 45 shows the syntax for declaring large object (LOB) host variables in FORTRAN.

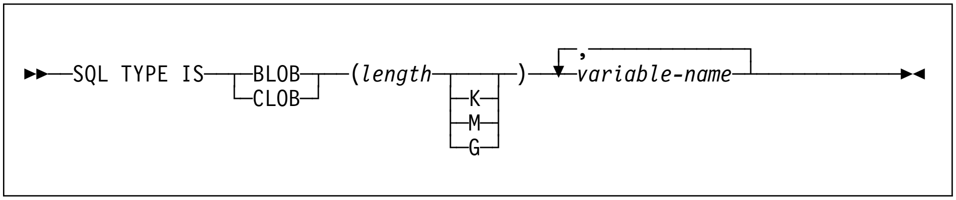


Figure 45. Syntax for Large Object (LOB) Host Variables in FORTRAN

Notes:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB, CLOB, K, M, G can be in either uppercase, lowercase, or mixed.
3. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
4. The initialization of a LOB within a LOB declaration is not permitted.
5. The host variable name prefixes 'length' and 'data' in the precompiler generated code.

BLOB Example:

Declaring:

```
sql type is blob(2m) my_blob
```

Results in the generation of the following structure:

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

CLOB Example:

Declaring:

```
sql type is clob(125m) my_clob
```

Results in the generation of the following structure:

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

LOB Locator Declarations in FORTRAN

Figure 46 shows the syntax for declaring large object (LOB) locator host variables in FORTRAN.

```
                                   ┌─,─────────────┐
►►──SQL TYPE IS──┬─BLOB_LOCATOR─┬──▼─variable-name─┴──────────────────────►◄
                 └─CLOB_LOCATOR─┘
```

Figure 46. Syntax for Large Object (LOB) Locator Host Variables in FORTRAN

Notes:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR can be either uppercase, lowercase, or mixed.
3. Initialization of locators is not permitted.

CLOB Locator Example (BLOB locator is similar):

Declaring:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

Results in the generation of the following declaration:

```
 integer*4 my_locator
```

File Reference Declarations in FORTRAN

Figure 47 shows the syntax for declaring file reference host variables in FORTRAN.

```
                                   ┌─,─────────────┐
►►──SQL TYPE IS──┬─BLOB_FILE─┬────▼─variable-name─┴──────────────────────────►◄
                 └─CLOB_FILE─┘
```

Figure 47. Syntax for File Reference Host Variables in FORTRAN

Notes:

1. Graphic types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_FILE, CLOB_FILE can be either uppercase, lowercase, or mixed.

Example of a BLOB file reference variable (CLOB file reference variable is similar):

```
     SQL TYPE IS BLOB_FILE my_file
```

Results in the generation of the following declaration:

```
      character     my_file(267)
      integer*4     my_file_name_length
      integer*4     my_file_data_length
      integer*4     my_file_file_options
      character*255 my_file_name
      equivalence(  my_file(1),
     +              my_file_name_length )
      equivalence(  my_file(5),
     +              my_file_data_length )
      equivalence(  my_file(9),
     +              my_file_file_options )
      equivalence(  my_file(13),
     +              my_file_name )
```

Supported SQL Data Types

Certain predefined FORTRAN data types correspond to database manager column types. Only these FORTRAN data types can be declared as host variables.

Table 18 shows the FORTRAN equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Table 18 (Page 1 of 3). SQL Data Types Mapped to FORTRAN Declarations

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
SMALLINT (500 or 501)	INTEGER*2	16-bit, signed integer

Table 18 (Page 2 of 3). SQL Data Types Mapped to FORTRAN Declarations

SQL Column Type[1]	**FORTRAN Data Type**	**SQL Column Type Description**
INTEGER (496 or 497)	INTEGER*4	32-bit, signed integer
REAL[2] (480 or 481)	REAL*4	Single precision floating point
DOUBLE[3] (480 or 481)	REAL*8	Double precision floating point
DECIMAL(*p*,*s*) (484 or 485)	No exact equivalent; use REAL*8	Packed decimal
CHAR(*n*) (452 or 453)	CHARACTER**n*	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	SQL TYPE IS VARCHAR(*n*) where *n* is from 1 to 4000	Variable-length character string
LONG VARCHAR (456 or 457)	SQL TYPE IS VARCHAR(*n*) where *n* is from 4001 to 32700	Long variable-length character string
CLOB(*n*) (408 or 409)	SQL TYPE IS CLOB (n) where *n* is from 1 to 2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	SQL TYPE IS CLOB_LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (808 or 809)	SQL TYPE IS CLOB_FILE	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	SQL TYPE IS BLOB(n) where *n* is from 1 to 2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	SQL TYPE IS BLOB_LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable[4] (804 or 805)	SQL TYPE IS BLOB_FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	CHARACTER*10	10-byte character string
TIME (388 or 389)	CHARACTER*8	8-byte character string
TIMESTAMP (392 or 393)	CHARACTER*26	26-byte character string

Table 18 (Page 3 of 3). SQL Data Types Mapped to FORTRAN Declarations

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description

Notes:

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following is a sample SQL declare section with a host variable declared for each supported data type:

```
EXEC SQL BEGIN DECLARE SECTION
  INTEGER*2     AGE  /26/
  INTEGER*4     DEPT
  REAL*4        BONUS
  REAL*8        SALARY
  CHARACTER     MI
  CHARACTER*112 ADDRESS
  SQL TYPE IS VARCHAR (512) DESCRIPTION
  SQL TYPE IS VARCHAR (32000) COMMENTS
  SQL TYPE IS CLOB (1M) CHAPTER
  SQL TYPE IS CLOB_LOCATOR CHAPLOC
  SQL TYPE IS CLOB_FILE  CHAPFL
  SQL TYPE IS BLOB (1M) VIDEO
  SQL TYPE IS BLOB_LOCATOR VIDLOC
  SQL TYPE IS BLOB_FILE VIDFL
  CHARACTER*10  DATE
  CHARACTER*8   TIME
  CHARACTER*26  TIMESTAMP
  INTEGER*2     WAGE_IND
EXEC SQL END DECLARE SECTION
```

The following are additional rules for supported FORTRAN data types:

- You may define dynamic SQL statements longer than 254 characters by using VARCHAR, LONG VARCHAR, OR CLOB host variables.

SQLSTATE and SQLCODE Variables

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables :

```
EXEC SQL BEGIN DECLARE SECTION;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

If neither of these is specified, the SQLCOD declaration is assumed during the precompile step. The variable named 'SQLSTATE' may also be 'SQLSTA'. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications that contain multiple source files, the declarations of SQLCOD and SQLSTATE may be included in each source file as shown above.

Considerations for Multi-byte Character Sets

There are no graphic (multi-byte) host variable data types supported in FORTRAN. Only mixed character host variables are supported through the `character` data type. It is possible to create a user SQLDA that contains graphic data.

Japanese or Traditional-Chinese EUC Considerations

Any graphic data sent from an application running under the Japanese or Traditional-Chinese EUC code set is tagged with the UCS-2 code set identifier so your application must convert a EUC string to UCS-2 before sending it to a database server. Similarly, graphic data retrieved from a database by an application running under an EUC code set is encoded using UCS-2, requiring that your application convert from UCS-2 to the EUC code set internally, unless the user is to be presented with UCS-2 data. The conversions to and from UCS-2 are your responsibility since this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. No application-accessible conversion routines are supplied with the DB2 products. Use system calls available from your operating system.

Character host variable data and character constants are tagged as mixed character data with the mixed EUC code page identifier.

For general EUC application development guidelines, see “Japanese and Traditional-Chinese EUC Code Set Considerations” on page 377.

Chapter 14. Programming in REXX

This chapter discusses host language specific information for DB2 applications. The following topics are addressed:

- Programming Considerations
- Execution Requirements for REXX
- API Syntax.

Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on embedding SQL statements, language restrictions, and supported data types for host variables.

Because REXX is an interpreted language, no precompiler, compiler, or linker is used. Instead, three DB2 APIs are used to create DB2 applications in REXX. Use these APIs to access different elements of DB2.

SQLEXEC Supports the SQL language

SQLDBS Supports command-like versions of DB2 APIs.

SQLDB2 Supports a REXX specific interface to the command-line processor. See “API Syntax” on page 501 for details and restrictions on how this interface can be used.

Language Restrictions

It is possible that tokens within statements or commands that are passed to the SQLEXEC, SQLDBS, and SQLDB2 routines could correspond to REXX variables. In this case, the REXX interpreter substitutes the variable’s value before calling SQLEXEC, SQLDBS, or SQLDB2.

To avoid this situation, enclose statement strings in quotes (' ' or " "). If you do not use quotes, any conflicting variable names are resolved by the REXX interpreter, instead of being passed to the SQLEXEC, SQLDBS or SQLDB2 routines.

Compound SQL is not supported in REXX/SQL.

REXX/SQL stored procedures are supported on the OS/2, Windows 95, and Windows NT operating systems, but not on AIX.

Registering SQLEXEC, SQLDBS and SQLDB2

Before using any of the DB2 APIs or issuing SQL statements in an application, you must register the SQLDBS, SQLDB2 and SQLEXEC routines. This notifies the REXX interpreter of the REXX/SQL entry points. The method you use for registering varies slightly between the OS/2 and AIX platforms. The following examples show the correct syntax for registering each routine:

Sample registration on OS/2 or Windows

```
/* ------------ Register SQLDBS with REXX  -------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy \= 0 then
    do
      say 'SQLDBS was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ------------ Register SQLDB2 with REXX  -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy \= 0 then
    do
      say 'SQLDB2 was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ----------------- Register SQLEXEC with REXX  --------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy \= 0 then
    do
      say 'SQLEXEC was not successfully added to the REXX environment'
      signal rxx_exit
    end
```

Sample registration on AIX

```
 /* ------------ Register SQLDBS, SQLDB2 and SQLEXEC with REXX --------*/
 rcy = SysAddFuncPkg("db2rexx")
 If rcy \= 0 then
   do
     say 'db2rexx was not successfully added to the REXX environment'
     signal rxx_exit
   end
```

On OS/2, the RxFuncAdd commands need to be executed only once for all sessions.

On AIX, the SysAddFuncPkg should be executed in every REXX/SQL application.

Details on the RXfuncadd and SysAddFuncPkg APIs are available in the REXX documentation for OS/2 and AIX, respectively.

Embedding SQL Statements

Use the SQLEXEC routine to process all SQL statements. The character string arguments for the SQLEXEC routine are made up of the following elements:

- SQL keywords
- Pre-declared identifiers
- Statement host variables.

Make each request by passing a valid SQL statement to the SQLEXEC routine. Use the following syntax:

```
CALL SQLEXEC 'statement'
```

SQL statements can be continued onto more than one line. Each part of the statement should be enclosed in single quotes, and a comma must delimit additional statement text as follows:

```
CALL SQLEXEC 'SQL text',
             'additional text',
                  .
                  .
                  .
             'final text'
```

The following is an example of embedding an SQL statement in REXX:

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

In this example, the `SQLCODE` field of the `SQLCA` structure is checked to determine whether the update was successful.

The following rules apply to embedded SQL statements:

- The following SQL statements can be passed directly to the SQLEXEC routine:

  CALL
  CLOSE
  COMMIT
  CONNECT
  CONNECT TO
  CONNECT RESET
  DECLARE
  DESCRIBE
  DISCONNECT
  EXECUTE
  EXECUTE IMMEDIATE
  FETCH
  FREE LOCATOR
  OPEN
  PREPARE
  RELEASE
  ROLLBACK
  SET CONNECTION

  Other SQL statements must be processed dynamically using the EXECUTE IMMEDIATE, or PREPARE and EXECUTE statements in conjunction with the SQLEXEC routine.

- You cannot use host variables in the CONNECT and SET CONNECTION statements in REXX.
- Cursor names and statement names are predefined as follows:

  c1 to c100 Cursor names, which range from *c1* to *c50* for cursors declared without the WITH HOLD option, and *c51* to *c100* for cursors declared using the WITH HOLD option.

  The cursor name identifier is used for DECLARE, OPEN, FETCH, and CLOSE statements. It identifies the cursor used in the SQL request.

  s1 to s100 Statement names, which range from *s1* to *s100*.

  The statement name identifier is used with the DECLARE, DESCRIBE, PREPARE, and EXECUTE statements.

  The pre-declared identifiers must be used for cursor and statement names. Other names are not allowed.
- When declaring cursors, the cursor name and the statement name should correspond in the DECLARE statement. For example, if *c1* is used as a cursor name, *s1* must be used for the statement name.
- Do not use comments within an SQL statement.

Host Variables

Host variables are REXX language variables that are referenced within SQL statements. They allow an application to pass input data to DB2 and receive output data from DB2. REXX applications do not need to declare host variables, except for LOB locators and LOB file reference variables. Host variable data types and sizes are determined at run time when the variables are referenced. Apply the following rules when naming and using host variables.

Naming Host Variables

Any properly named REXX variable can be used as a host variable. A variable name can be up to 64 characters long. Do not end the name with a period. A host variable name can consist of alphabetic characters, numerics, and the characters @, _, !, ., ?, and $.

Referencing Host Variables

The REXX interpreter examines every string without quotes in a procedure. If the string represents a variable in the current REXX variable pool, REXX replaces the string with the current value. The following is an example of how you can reference a host variable in REXX:

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

To ensure that a character string is not converted to a numeric data type, enclose the string with single quotes as in the following example:

```
VAR = '100'
```

REXX sets the variable *VAR* to the 3-byte character string `100`. If single quotes are to be included as part of the string, follow this example:

```
VAR = "'100'"
```

When inserting numeric data into a CHARACTER field, the REXX interpreter treats numeric data as integer data, thus you must concatenate numeric strings explicitly and surround them with single quotes.

Indicator Variables in REXX

An indicator variable data type in REXX is a number without a decimal point. Following is an example of an indicator variable in REXX using the INDICATOR keyword.

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

In the above example, `cmind` is examined for a negative value. If it is not negative, the application can use the returned value of `cm`. If it is negative, the fetched value is NULL and `cm` should not be used. The database manager does not change the value of the host variable in this case.

Predefined REXX Variables

SQLEXEC, SQLDBS and SQLDB2 set predefined REXX variables as a result of certain operations. These variables are:

RESULT Each operation sets this return code. Possible values are:

n Where *n* is a positive value indicating the number of bytes in a formatted message. The GET ERROR MESSAGE API alone returns this value.

0 The API was executed. The REXX variable SQLCA contains the completion status of the API. If SQLCA.SQLCODE is not zero, SQLMSG contains the text message associated with that value.

–1 There is not enough memory available to complete the API. The requested message was not returned.

–2 SQLCA.SQLCODE is set to 0. No message was returned.

–3 SQLCA.SQLCODE contained an invalid SQLCODE. No message was returned.

–6 The SQLCA REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.

–7 The SQLMSG REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.

–8 The SQLCA.SQLCODE REXX variable could not be fetched from the REXX variable pool.

–9 The SQLCA.SQLCODE REXX variable was truncated during the fetch. The maximum length for this variable is 5 bytes.

–10 The SQLCA.SQLCODE REXX variable could not be converted from ASCII to a valid long integer.

–11 The SQLCA.SQLERRML REXX variable could not be fetched from the REXX variable pool.

–12 The SQLCA.SQLERRML REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.

–13 The SQLCA.SQLERRML REXX variable could not be converted from ASCII to a valid short integer.

–14 The SQLCA.SQLERRMC REXX variable could not be fetched from the REXX variable pool.

–15 The SQLCA.SQLERRMC REXX variable was truncated during the fetch. The maximum length for this variable is 70 bytes.

–16 The REXX variable specified for the error text could not be set.

–17 The SQLCA.SQLSTATE REXX variable could not be fetched from the REXX variable pool.

–18 The SQLCA.SQLSTATE REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.

Note: The values –8 through –18 are returned only by the GET ERROR MESSAGE API.

SQLMSG If SQLCA.SQLCODE is not 0, this variable contains the text message associated with the error code.

SQLISL The isolation level. Possible values are:

RR Repeatable read.

RS Read stability.

CS Cursor stability. This is the default.

UR Uncommitted read.

NC No commit (NC is only supported by some DRDA servers.)

SQLCA The SQLCA structure updated after SQL statements are processed and DB2 APIs are called. The entries of this structure are described in the *API Reference*.

SQLRODA The input/output SQLDA structure for stored procedures invoked using the CALL statement. It is also the output SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *API Reference*.

SQLRIDA The input SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *API Reference*.

SQLRDAT An SQLCHAR structure for server procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *API Reference*.

LOB Host Variables in REXX

When you fetch a LOB column into a REXX host variable, it will be stored as a simple (that is, uncounted) string. This is handled in the same manner as all character-based

SQL types (such as CHAR, VARCHAR, GRAPHIC, LONG, and so on). On input, if the size of the contents of your host variable is larger than 32K, or if it meets other criteria set out below, it will be assigned the appropriate LOB type.

In REXX SQL, LOB types are determined from the string content of your host variable as follows:

Host variable string content	**Resulting LOB type**
:hv1='ordinary quoted string longer than 32K ...'	CLOB
:hv2="'string with embedded delimiting quotes "", "longer than 32K...'"	CLOB
:hv3="G'DBCS string with embedded delimiting single "", "quotes, beginning with G, longer than 32K...'"	DBCLOB
:hv4="BIN'string with embedded delimiting single "", "quotes, beginning with BIN, any length...'"	BLOB

LOB Locator Declarations in REXX

Figure 48 shows the syntax for declaring LOB locator host variables in REXX.

```
             ┌─,─────────────────┐
►►──DECLARE──▼─:──variable-name──┴──LANGUAGE TYPE──┬─BLOB───┬──LOCATOR──►◄
                                                   ├─CLOB───┤
                                                   └─DBCLOB─┘
```

Figure 48. Syntax for LOB Locator Host Variables in REXX

You must declare LOB locator host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as locators for the remainder of the program. Locator values are stored in REXX variables in an internal format.

Example:

```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

Data represented by LOB locators returned from the engine can be freed in REXX/SQL using the FREE LOCATOR statement which has the following format:

```
                      ┌─,─────────────────┐
►►──FREE──LOCATOR──▼─:──variable-name──┴──────────────────────────────►◄
```

Figure 49. Syntax for FREE LOCATOR statement in REXX

Example:

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

LOB File Reference Declarations in REXX

You must declare LOB file reference host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as LOB file references for the remainder of the program.

Figure 50 shows the syntax for declaring LOB file reference host variables in REXX.

```
              ┌─,──────────────────┐
►►──DECLARE───▼─:──variable-name───┴───LANGUAGE TYPE──┬─BLOB───┬──FILE───────►◄
                                                      ├─CLOB───┤
                                                      └─DBCLOB─┘
```

Figure 50. Syntax for LOB File Reference Variables in REXX

Example:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

File reference variables in REXX contain three fields. For the above example they are:

`hv3.FILE_OPTIONS.` Set by the application to indicate how the file will be used.
`hv3.DATA_LENGTH.` Set by DB2 to indicate the size of the file.
`hv3.NAME.` Set by the application to the name of the LOB file.

For FILE_OPTIONS, the application sets the following keywords:

Keyword (Integer Value) **Meaning**

READ (2) File is to be used for input. This is a regular file that can be opened, read and closed. The length of the data in the file (in bytes) is computed (by the application requestor code) upon opening the file.

CREATE (8) On output, create a new file. If the file already exists, it is an error. The length (in bytes) of the file is returned in the `DATA_LENGTH` field of the file reference variable structure.

OVERWRITE (16) On output, the existing file is overwritten if it exists, otherwise a new file is created. The length (in bytes) of the file is returned in the `DATA_LENGTH` field of the file reference variable structure.

APPEND (32) The output is appended to the file if it exists, otherwise a new file is created. The length (in bytes) of the data that was added to the file (not the total file length) is returned in the `DATA_LENGTH` field of the file reference variable structure.

Note: A file reference host variable is a compound variable in REXX, thus you must set values for the `NAME`, `NAME_LENGTH` and `FILE_OPTIONS` fields in addition to declaring them.

Clearing LOB Host Variables

On OS/2 it may be necessary to explicitly clear REXX SQL LOB locator and file reference host variable declarations as they remain in effect after your application program ends. This is because the application process does not exit until the session in which it is run is closed. If REXX SQL LOB declarations are not cleared, they may interfere with other applications that are running in the same session after a LOB application has been executed.

The syntax to clear the declaration is:

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

You should code this statement at the end of LOB applications. Note that you can code it anywhere as a precautionary measure to clear declarations which might have been left by previous applications (for example, at the beginning of a REXX SQL application).

Supported SQL Data Types

Certain predefined REXX data types correspond to DB2 column types. Table 19 shows how SQLEXEC and SQLDBS interpret REXX variables in order to convert their contents to DB2 data types.

Table 19 (Page 1 of 3). SQL Column Types Mapped to REXX Declarations

SQL Column Type[1]	**REXX Data Type**	**SQL Column Type Description**
SMALLINT (500 or 501)	A number without a decimal point ranging from -32 768 to 32 767	16-bit signed integer
INTEGER (496 or 497)	A number without a decimal point ranging from -2 147 483 648 to 2 147 483 647	32-bit signed integer
REAL[2] (480 or 481)	A number in scientific notation ranging from -3.40282346 x 10^{38} to 3.40282346 x 10^{38}	Single-precision floating point
DOUBLE[3] (480 or 481)	A number in scientific notation ranging from -1.79769313 x 10^{308} to 1.79769313 x 10^{308}	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	A number with a decimal point	Packed decimal

Table 19 (Page 2 of 3). SQL Column Types Mapped to REXX Declarations

SQL Column Type[1]	**REXX Data Type**	**SQL Column Type Description**
CHAR(*n*) (452 or 453)	A string with a leading and trailing quote ('), which has length *n* after removing the two quote marks A string of length *n* with any non-numeric characters, other than leading and trailing blanks or the E in scientific notation	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	Equivalent to CHAR(*n*)	Variable-length character string of length *n*, where *n* ranges from 1 to 4000
LONG VARCHAR (456 or 457)	Equivalent to CHAR(*n*)	Variable-length character string of length *n*, where *n*ranges from 1 to 32 700
CLOB(*n*) (408 or 409)	Equivalent to CHAR(*n*)	Large object variable-length character string of length *n*, where *n* ranges from 1 to 2 147 483 647
CLOB locator variable[4] (964 or 965)	DECLARE *:var_name* LANGUAGE TYPE CLOB LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (808 or 809)	DECLARE *:var_name* LANGUAGE TYPE CLOB FILE	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	A string with a leading and trailing apostrophe, preceded by BIN, containing *n* characters after removing the preceding BIN and the two apostrophes.	Large object variable-length binary string of length *n*, where *n* ranges from 1 to 2 147 483 647
BLOB locator variable[4] (960 or 961)	DECLARE *:var_name* LANGUAGE TYPE BLOB LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable[4] (804 or 805)	DECLARE *:var_name* LANGUAGE TYPE BLOB FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	Equivalent to CHAR(*10*)	10-byte character string
TIME (388 or 389)	Equivalent to CHAR(*8*)	8-byte character string
TIMESTAMP (392 or 393)	Equivalent to CHAR(*26*)	26-byte character string

Table 19 (Page 3 of 3). SQL Column Types Mapped to REXX Declarations

SQL Column Type[1]	**REXX Data Type**	**SQL Column Type Description**
Note: The following data types are only available in the DBCS environment.		
GRAPHIC(*n*) (468 or 469)	A string with a leading and trailing apostrophe preceded by a G or N, containing *n* DBCS characters after removing the preceding character and the two apostrophes	Fixed-length graphic string of length *n*, where *n* is from 1 to 127
VARGRAPHIC(*n*) (464 or 465)	Equivalent to GRAPHIC(*n*)	Variable-length graphic string of length *n*, where *n* ranges from 1 to 2000
LONG VARGRAPHIC (472 or 473)	Equivalent to GRAPHIC(*n*)	Long variable-length graphic string of length *n*, where *n* ranges from 1 to 16 350
DBCLOB(*n*) (412 or 413)	Equivalent to GRAPHIC(*n*)	Large object variable-length graphic string of length *n*, where *n* ranges from 1 to 1 073 741 823
DBCLOB locator variable[4] (968 or 969)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB LOCATOR	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (812 or 813)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB FILE	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string.
2. FLOAT(*n*) where 0 < *n* < 25 is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where 24 < *n* < 54 is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

Using Cursors in REXX

When a cursor is declared in REXX, the cursor is associated with a query. The query is associated with a statement name assigned in the PREPARE statement. Any referenced host variables are represented by parameter markers. The following example shows a DECLARE statement associated with a dynamic SELECT statement.

```
prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"
CALL SQLEXEC 'PREPARE S1 FROM :prep_string';
CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';
CALL SQLEXEC 'OPEN C1 USING :schema_name';
```

Execution Requirements for REXX

REXX applications are not precompiled, compiled, or linked.

On OS/2, your application file must have a .CMD extension. After creation, you can run your application directly from the operating system command prompt.

On the Windows 95 or Windows NT operating system, your application file can have any name. After creation, you can run your application from the operating system command prompt by invoking the REXX interpreter as follows:

```
REXX file_name
```

On AIX, your application file can have any extension. You can run your application using either of the following two methods:

1. At the shell command prompt, type `rexx name` where `name` is the name of your REXX program.
2. If the first line of your REXX program contains a "magic number" (#!) and identifies the directory where the REXX/6000 interpreter resides, you can run your REXX program by typing its name at the shell command prompt. For example, if the REXX/6000 interpreter file is in the /usr/bin directory, include the following as the very first line of your REXX program:

   ```
   #! /usr/bin/rexx
   ```

   Then, make the program executable by typing the following command at the shell command prompt:

   ```
   chmod +x name
   ```

   Run your REXX program by typing its file name at the shell command prompt.

Note: On AIX, you should set the LIBPATH environment variable to include the directory where the REXX SQL library, `db2rexx` is located. For example:

```
export LIBPATH=/lib:/usr/lib:/usr/lpp/db2_05_00/lib
```

Bind Files for REXX

Five bind files are provided to support REXX applications. The names of these files are included in the DB2UBIND.LST file. Each bind file was precompiled using a different isolation level; therefore, there are five different packages stored in the database.

The five bind files are:

DB2ARXCS.BND
Supports the cursor stability isolation level

DB2ARXRR.BND
Supports the repeatable read isolation level

DB2ARXUR.BND
Supports the uncommitted read isolation level

DB2ARXRS.BND
Supports the read stability isolation level.

DB2ARXNC.BND
Supports the no commit isolation level. This isolation level is used when working with some DRDA database servers. On other databases, it behaves like the uncommitted read isolation level.

Note: In some cases, it may be necessary to explicitly bind these files to the database.

When you use the SQLEXEC routine, the package created with cursor stability is used as a default. If you require one of the other isolation levels, you can change isolation levels with the SQLDBS CHANGE SQL ISOLATION LEVEL API, before connecting to the database. This will cause subsequent calls to the SQLEXEC routine to be associated with the specified isolation level.

OS/2 REXX applications cannot assume that the default isolation level is in effect unless they know that no other REXX programs in the session have changed the setting. Before connecting to a database, a REXX application should explicitly set the isolation level.

API Syntax

Use the SQLDBS routine to call DB2 APIs with the following syntax:

```
CALL SQLDBS 'command string'
```

For information on how the DB2 APIs work, see the complete descriptions in the DB2 API chapter of the *API Reference*.

If a DB2 API you want to use cannot be called using the SQLDBS routine (and consequently, not listed in the *API Reference*), you may still call the API by calling the DB2 command line processor (CLP) from within the REXX application. However, since the DB2 CLP directs output either to the standard output device or to a specified file, your REXX application cannot directly access the output from the called DB2 API nor can it easily make a determination as to whether the called API is successful or not. The SQLDB2 API provides an interface to the DB2 CLP that provides direct feedback to your REXX application on the success or failure of each called API by setting the compound REXX variable, SQLCA, after each call.

You can use the SQLDB2 routine to call DB2 APIs using the following syntax:

```
CALL SQLDB2 'command string'
```

where `'command string'` is a string that can be processed by the command-line processor (CLP). See the *Command Reference* for the syntax of strings that can be processed by the CLP.

Calling a DB2 API using SQLDB2 is equivalent to calling the CLP directly, except for the following:

- The call to the CLP executable is replaced by the call to SQLDB2 (all other CLP options and parameters are specified the same way).
- The REXX compound variable SQLCA is set after calling the SQLDB2 but is not set after calling the CLP executable.
- The default display output of the CLP is set to off when you call SQLDB2, whereas the display is set to on output when you call the CLP executable. Note that you can turn the display output of the CLP to on by passing the `+o` or the `−o−` option to the SQLDB2.

Since the only REXX variable that is set after you call SQLDB2 is the SQLCA, you only use this routine to call DB2 APIs that do not return any data other than the SQLCA and that are not currently implemented through the SQLDBS interface. Thus, only the following DB2 APIs are supported by SQLDB2:

Activate Database
Add Node
Bind for DB2 Version 1[7], [8]
Bind for DB2 Version 2 or 3[7]
Create Database at Node
Drop Database at Node
Drop Node Verify
Deactivate Database
Deregister
Load[9]
Load Query
Precompile Program[7]
Rebind Package[7]
Redistribute Nodegroup
Register
Start Database Manager
Stop Database Manager

[7] These commands require a CONNECT statement through the SQLDB2 interface. Connections using the SQLDB2 interface are not accessible to the SQLEXEC interface and connections using the SQLEXEC interface are not accessible to the SQLDB2 interface.

[8] Is supported on OS/2 through the SQLDB2 interface.

[9] The optional output parameter, `pLoadInfoOut` for the Load API is not returned to the application in REXX. Refer to the *API Reference* for more information on the Load API and its parameters.

Note: Although the SQLDB2 routine is intended to be used only for the DB2 APIs listed above, it can also be used for other DB2 APIs that are not supported through the SQLDBS routine. Alternatively, the DB2 APIs can be accessed through the CLP from within the REXX application.

REXX Stored Procedures

REXX SQL applications can call stored procedures at the database server by using the SQL CALL statement. The stored procedure can be written in any language supported on that server, except for REXX on AIX systems. (Client applications may be written in REXX on AIX systems, but, as with other languages, they cannot call a stored procedure written in REXX on AIX.)

Calling Stored Procedures

The CALL statement allows a client application to pass data to, and receive data from, a server stored procedure. The interface for both input and output data is a list of host variables (refer to the *SQL Reference* for details). Because REXX generally determines the type and size of host variables based on their content, any output-only variables passed to CALL should be initialized with *dummy* data similar in type and size to the expected output.

Data can also be passed to stored procedures through SQLDA REXX variables, using the USING DESCRIPTOR syntax of the CALL statement. Table 20 shows how the SQLDA is set up. In the table, ':value' is the stem of a REXX host variable that contains the values needed for the application. For the DESCRIPTOR, 'n' is a numeric value indicating a specific *sqlvar* element of the SQLDA. The numbers on the right refer to the notes following Table 20.

Table 20. Client-side REXX SQLDA for Stored Procedures using the CALL Statement

USING DESCRIPTOR	:value.SQLD	1	
	:value.n.SQLTYPE	1	
	:value.n.SQLLEN	1	
	:value.n.SQLDATA	1	2
	:value.n.SQLDIND	1	2

Notes:

1. Before invoking the stored procedure, the client application must initialize the REXX variable with appropriate data.

   When the SQL CALL statement is executed, the database manager allocates storage and retrieves the value of the REXX variable from the REXX variable pool. For an SQLDA used in a CALL statement, the database manager allocates storage for the SQLDATA and SQLIND fields based on the SQLTYPE and SQLLEN values.

   In the case of a REXX stored procedure (that is, the procedure being called is itself written in OS/2 REXX), the data passed by the client from either type of CALL statement or the DARI API is placed in the REXX variable pool at the database server using the following predefined names:

SQLRIDA Predefined name for the REXX input SQLDA variable

SQLRODA Predefined name for the REXX output SQLDA variable

2. When the stored procedure terminates, the database manager also retrieves the value of the variables from the stored procedure. The values are returned to the client application and placed in the client's REXX variable pool.

Considerations on the Client

When using host variables in the CALL statement, initialize each host variable to a value that is type compatible with any data that is returned to the host variable from the server procedure. You should perform this initialization even if the corresponding indicator is negative.

When using descriptors, SQLDATA must be initialized and contain data that is type compatible with any data that is returned from the server procedure. You should perform this initialization even if the SQLIND field contains a negative value. See "REXX Example: OUTCLI.CMD" on page 191 for an example of this initialization.

Considerations on the Server

Ensure that all the SQLDATA fields and SQLIND (if it is a nullable type) of the predefined output sqlda SQLRODA are initialized. For example, if SQLRODA.SQLD is 2, the following fields must contain some data (even if the corresponding indicators are negative and the data is not passed back to the client):

- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

Retrieving Precision and SCALE Values from SQLDA Decimal Fields

To retrieve the precision and scale values for decimal fields from the SQLDA structure returned by the database manager, use the `sqllen.scale` and `sqllen.precision` values when you initialize the SQLDA output in your REXX program. For example, the following code fragment is taken from the "REXX Example: OUTCLI.CMD" on page 191:

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */   4
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485           /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale  = 2       /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision  = 7   /* WIDTH OF DECIMAL  */
io_sqlda.1.sqldata = 00000.00      /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1             /* NO INPUT DATA */
.
.
.
```

Japanese or Traditional-Chinese EUC Considerations

REXX applications are not supported under Japanese or Traditional-Chinese EUC environments.

REXX Sample Programs

The following is a list of the supplied example programs that are written in REXX. For the details of how the programs work, see the descriptions preceding each example:

- “REXX Example: UPDAT.CMD” on page 81
- “REXX Example: DYNAMIC.CMD” on page 101
- “REXX Example: OUTCLI.CMD” on page 191
- “REXX OS/2 Example: OUTSRV.CMD” on page 199
- “REXX Example: INPCLI.CMD” on page 213
- “REXX OS/2 Example: INPSRV.CMD” on page 225

Chapter 15. Programming in Java

If you want to use Java as a programming language to access data from your DB2 databases, DB2 provides support for:

- Developing Java applications and applets that access DB2 databases through the Java Development Kit (JDK) Version 1.1. The JDK includes the Java Database Connectivity (JDBC) API from Sun Microsystems. See the Web Page at "`http://www.software.ibm.com/data/db2/java`" for information on this specification.
- Creating user-defined functions (UDFs) and stored procedures in Java.

This chapter describes how to access DB2 databases using the Java programming language. The specific topics discussed are:

- Getting Started
- How Does It Work?
- Creating and Running JDBC Applets and Applications
- Creating Java UDFs and Stored Procedures

Getting Started

You can use DB2's JDBC support to build both:

- Java applications, which rely on the DB2 Client Application Enabler (CAE) to connect to DB2.
- Java applets, that do not require any DB2 component code on the client.

If your application or applet uses JDBC, you need to familiarize yourself with the JDBC specification, *JDBC: A Java SQL API* available from Sun Microsystems. This specification describes how to call JDBC APIs to access a database and manipulate data in that database.

You should also read through this section to learn about DB2's extensions to JDBC and its few limitations (refer to "Extensions" on page 511). If you plan to create UDFs or stored procedures in Java, refer to "Creating and Using Java User-Defined Functions" on page 514 and "Creating and Using Java Stored Procedures" on page 516, as there are considerations that are different for Java than for other languages.

To help you begin coding your program, sample applications and applets are provided in the `sqllib/samples/java` directory. If you have created the SAMPLE database, you can also run the samples. See the *SQL Reference* for your platform for information about the SAMPLE database.

How Does It Work?

There are two independent components to DB2's Java enablement:

- Support for client applications and applets written in Java using JDBC to access DB2 (see "JDBC Applet and Application Support").
- Support for Java UDFs and stored procedures on the server (see "Java UDFs and Stored Procedures" on page 509)

JDBC Applet and Application Support

Figure 51 illustrates how the JDBC applet driver works. The driver consists of a JDBC client and a JDBC server. The JDBC client driver is loaded on the Web browser along with the applet. When a connection to a DB2 database is requested by the applet, the client opens a TCP/IP socket to the JDBC server on the machine where the Web server is running. After a connection is set up, the client sends each of the subsequent database access requests from the applet to the JDBC server though the TCP/IP connection. The JDBC server then makes corresponding CLI (ODBC) calls to perform the task. Upon completion, the JDBC server sends the results back to the client through the connection.

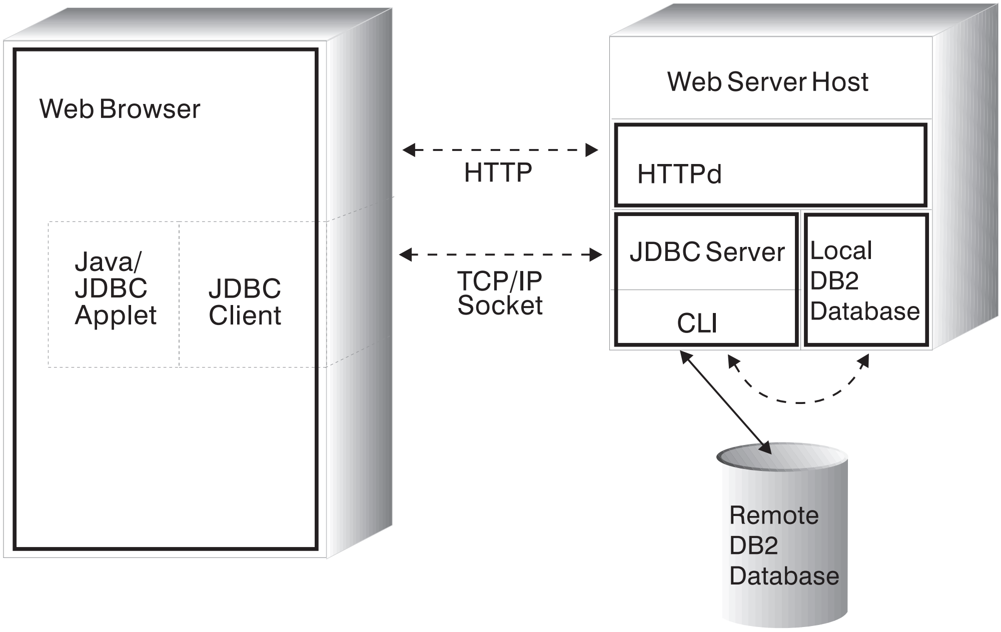


Figure 51. DB2's JDBC Applet Implementation

Figure 52 on page 509 illustrates how a DB2 JDBC application works. You can think of a DB2 JDBC application as a DB2 CLI application, only you write it using the Java language. Calls to JDBC are translated to calls to DB2 CLI, through Java native methods. This dependency requires that the DB2 CAE component be installed at the client. A JDBC request flows through DB2 CLI to the DB2 server through the normal CAE communication flow.

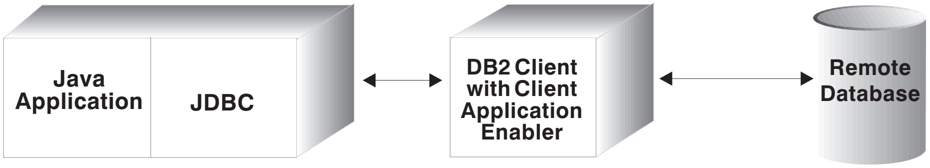


Figure 52. DB2's JDBC Application Implementation

Writing a Java JDBC application or applet is very similar to writing a C application using DB2 CLI or ODBC to access the database. The primary difference between applications and applets is that an application requires the DB2 CAE to be installed on the client, and uses the CAE to communicate with DB2; the applet depends on a Java-enabled Web browser, and does not require any DB2 code installed on the client.

You need to treat applets differently than applications, as applets are delivered over the World Wide Web (WWW). You must install the DB2 server or client on the same machine as your Web server. The JDBC applet and application support is installed as part of DB2.

Java UDFs and Stored Procedures

Creating Java UDFs and stored procedures is very similar to creating UDFs and stored procedures in other supported programming languages. Once you have created and registered them, you can call them from programs in any language. Typically, you may call JDBC APIs from your stored procedures, but you can not call them from UDFs.

To run your UDFs and stored procedures, DB2 calls the Java interpreter on the server. DB2 does not include a Java interpreter; your database administrator must install and configure the appropriate Java Development Kit (JDK) on your DB2 server before starting up the database.

The runtime libraries for the Java interpreter must be available in the system search paths (PATH or LIBPATH or LD_LIBRARY_PATH, and CLASSPATH). For more information on setting up your environment for Java, see the *DB2 SDK Building Applications* book for your operating system.

DB2 loads or starts the Java interpreter on the first call to a Java UDF or stored procedure (see “Creating and Using Java User-Defined Functions” on page 514 or “Creating and Using Java Stored Procedures” on page 516). For unfenced UDFs and stored procedures, DB2 loads one Java interpreter per database instance, and runs it inside the database engine's address space for best performance. For fenced UDFs, DB2 uses a distinct interpreter inside the db2udf process; similarly, fenced stored procedures use a distinct interpreter inside the db2dari process. In all cases, the Java interpreter stays loaded until the embedding process ends.

Creating and Running JDBC Applets and Applications

Whether you're writing an application or applet, you would typically call JDBC APIs to:

1. Import the appropriate Java packages and classes (`java.sql.*`).
2. Load the appropriate JDBC driver (`COM.ibm.db2.jdbc.app.DB2Driver` for applications; `COM.ibm.db2.jdbc.net.DB2Driver` for applets).
3. Connect to the database, specifying the location with a URL (as defined in Sun's JDBC specification) and using the `db2` subprotocol. For applets, you must also provide the userid, password, host name, and the port number for the applet server; for applications, the Client Application Enabler provides the required values.
4. Pass SQL statements to the database.
5. Receive the results.
6. Close the connection.

After coding your program, compile it as you would any other Java program. You don't need to perform any special precompile or bind steps.

Distributing and Running a JDBC Applet

Like other Java applets, you distribute your JDBC applet over the World Wide Web (WWW). Typically you would imbed the applet in a hypertext markup language (HTML) page. For example, to call the sample applet `DB2Applt.java`, (provided in `sqllib/samples/java`) you might use the tag:

```
<applet code="DB2Applt.class" width=325 height=275 archive="db2java.zip">
```

To run your applet, you need only a Java-enabled Web browser on the client machine. When you load your HTML page, the applet tag down loads the Java applet to your machine which then downloads the Java class files, including the `COM.ibm.db2.jdbc.net` class which is DB2's JDBC driver. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes separate communications with the DB2 database through the JDBC applet server residing on the Web server.

For your applets to run, you need to ensure that the correct files are installed in the proper places, as follows:

1. Install DB2 (server or client) on the same machine as your Web server. The Java applet and JDBC support is installed as part of DB2.
2. Create your own HTML file, or use a modified version of the supplied `DB2Applt.html` file and move it to the same directory as your class file. Copy your `.class` file, or for the sample, `sqllib/samples/java/samples.zip` and the `sqllib/java/db2java.zip` file to this directory.
3. Start DB2's JDBC applet server on your Web server by typing:

   ```
   db2jstrt portno
   ```

   where *portno* is an unused TCP/IP port number that applets can use.

4. On your client machine, start your Web browser and load your HTML file to run your applet.

Distributing and Running a JDBC Application

Distribute your JDBC application as you would any other Java application. As the application uses DB2's CAE to communicate with the DB2 server, you have no special security concerns; authority verification is performed by the CAE.

To run your application on a client machine, you must also have installed on that machine:

- The Java interpreter, which you need to run any Java code.
- DB2's Client Application Enabler, which also includes the DB2 JDBC driver.

Start your application from the GUI or command line, like any other application.

A sample application, DB2Appl.java, is provided in the `sqllib/samples/java` directory. If you have created the SAMPLE database, you can also run the sample by adding `samples/java/samples.zip` to your CLASSPATH environment variable, changing to the `sqllib/samples/java` directory, and entering the following command:

```
java DB2Appl
```

See *SQL Reference* for information on the SAMPLE database.

Extensions

DB2 Universal Database supports the JDK Version 1.1 JDBC specification. You may have to modify applications and applets that are written to the JDBC Version 1.0 to work properly with DB2 Version 5.

There are also special considerations for graphical and large objects (LOBs).

Using Graphical and Large Objects

The JDBC specification does not explicitly mention large objects (LOBs) or graphic types.

Treat LOBs as the corresponding LONGVAR type. Because LOB types are declared in SQL with a maximum length, ensure that you do not return arrays or strings longer than the declared limit. This consideration applies to SQL string types as well.

For GRAPHIC and DBCLOB types, treat them as the corresponding CHAR types. The following JDBC APIs behave as described below:

setString Converts from Unicode to database format.[1]

setAsciiStream Converts from local code page to database format.[2]

setUnicodeStream Converts from Unicode to database format.[1]

getString Converts from database format to Unicode.[1]

getAsciiStream Converts from database format to local code page format.[2]

getUnicodeStream Converts from database format to Unicode.[1]

Notes:

1. The DB2 client first converts the data from the database format to the client format; DB2 then converts the data to Unicode
2. The DB2 client performs this type conversion. See your DB2 client information for supported code pages and conversions.

Creating Java UDFs and Stored Procedures

Along with supporting client-side Java code, DB2 also supports creating user-defined functions (UDF) and stored procedures in Java that reside on the server. This Java support does not alter the support for UDFs and stored procedures in other programming languages. For complete details on DB2's stored procedure and UDF support, see Chapter 5, "Writing Stored Procedures" on page 167, and Chapter 7, "Writing User-Defined Functions (UDFs)" on page 285.

UDFs and stored procedures written in Java provide the same capability as existing UDFs and stored procedures; they are simply methods in Java classes. Once you create and register these UDFs and stored procedures, and place the Java classes in the correct file location, described in "Where to Put Java Classes" on page 514. You can then call them from a program in any language. DB2 calls the Java interpreter to run them; they run as if part of a Java application, and therefore are not subject to applet security restrictions.

DB2 handles type conversion (see "Mapping Between SQL Types and Java Objects") between SQL types and Java objects for you, as it does for other programming languages. Because SQL string and LOB types are declared in SQL with a maximum length, ensure that your Java methods do not return arrays or strings that are longer than the declared limit. DB2 detects many possible errors in data conversion and signals them by throwing an exception.

Because you can overload Java methods, two methods with the same name but different argument lists can coexist in the same Java class. Make sure that your Java methods that implement UDFs and stored procedures have the exact *signature* expected, that is, the list of formal arguments and the method name.

Mapping Between SQL Types and Java Objects

When you call UDFs and stored procedures that are implemented as Java methods, DB2 converts SQL types to and from Java types for you as described in Table 21 on page 513. Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 21. DB2 SQL Types and Java Objects

SQL Type	Java Type (UDF)	Java Type (Stored Procedure)
SMALLINT (500/501)	short	short
INTEGER (496/497)	Int	Int
FLOAT (480/481)	double	double
REAL (480/481)[1]	float	float
DECIMAL(p,s) (484/485)	BigDecimal	BigDecimal
NUMERIC(p,s) (504/505)	BigDecimal	BigDecimal
CHAR(*n*) (452/453)	String	String
CHAR(*n*) FOR BIT DATA (452/453)	Blob	Blob
C null-terminated string (400/401)[2]	n/a	String
VARCHAR(*n*)(448/449)	String	String
VARCHAR(*n*) FOR BIT DATA (448/449)	Blob	Blob
LONG VARCHAR (456/457)	Clob	Clob
LONG VARCHAR FOR BIT DATA (456/457)	Blob	Blob
GRAPHIC(*n*) (468/469)	String	String
C null-terminated graphic string (460/461)[2]	n/a	String
VARGRAPHIC(*n*) (464/465)	String	String
LONG VARGRAPHIC (472/473)[3]	Clob	Clob
BLOB(*n*)(404/405)[3]	Blob	Blob
CLOB(*n*) (408/409)[3]	Clob	Clob
DBCLOB(*n*) (412/413)[3]	Clob	Clob
DATE (384/385)[4]	String	String
TIME (388/389)[4]	String	String
TIMESTAMP (392/393)[4]	String	String

Notes:

1. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
2. Parenthesized types, such as the C null-terminated graphic string, occur in stored procedures when the calling application uses embedded SQL with some host variable types.
3. The Blob and Clob classes are provided in the `COM.ibm.db2.app` package. Their interfaces include routines to generate an InputStream and OutputStream for reading from and writing to a Blob, and a Reader and Writer for a Clob. Refer to “Classes for Java Stored Procedures and UDFs” on page 518 for descriptions of the classes.
4. SQL DATE, TIME, and TIMESTAMP values use the ISO string encoding in Java, as they do for UDFs coded in C. This encoding is documented in Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285.

Instances of classes COM.ibm.db2.app.Blob and COM.ibm.db2.app.Clob represent the LOB data types (BLOB, CLOB, and DBCLOB). These classes provide a limited interface to read LOBs passed as inputs, and write LOBs returned as outputs. Reading and writing of LOBs occur through standard Java I/O stream objects. For the Blob class, the routines getInputStream() and getOutputStream() return an InputStream or OutputStream object through which the BLOB content may be processed bytes-at-a-time. For a Clob, the routines getReader() and getWriter() will return a Reader or Writer object through which the CLOB or DBCLOB content may be processed characters-at-a-time.

If such an object is returned as an output using the set() method, code page conversions may be applied in order to represent the Java Unicode characters in the database code page.

Where to Put Java Classes

Store all Java class files that implement UDFs or stored procedures into the sqllib/function directory. If you declare a class to be part of a Java package, create the corresponding subdirectories under sqllib/function and place the files in the correct subdirectory. For example, if you create a class ibm.tests.test1, store the corresponding Java byte-code file (named test1.class) in sqllib/function/ibm/tests.

The Java interpreter that DB2 invokes uses the CLASSPATH environment variable to locate Java files. DB2 adds the entries sqllib/function and sqllib/java/db2java.zip to the front of your CLASSPATH setting.

To set your environment so that the Java interpreter can find where you have stored the Java class files may need to set the jdk11_path configuration parameter, or else use the default value. Also, you may need to set the java_heap_sz configuration parameter to increase the heap size for your application. See the *Administration Guide* for more information on these configuration parameters.

Creating and Using Java User-Defined Functions

You can create and use UDFs in Java just as you would in other languages, with only a few minor differences. After you code the UDF, you register it with the database using the CREATE FUNCTION statement. See the *SQL Reference* for information on registering a Java UDF using this statement. You can then call it from any DB2 application in any language, including Java. The UDF can be fenced or unfenced, and you can also use options to modify how the UDF is run. See “Changing How a Java UDF Runs” on page 516.

Some sample Java UDFs are provided in DB2Udf.java in the sqllib/samples/java directory. To register and invoke the sample UDFs, follow the instructions in the DB2Udf.java file. For general details on DB2’s UDF support, see Chapter 7, “Writing User-Defined Functions (UDFs)” on page 285.

Coding a Java UDF

In general, if you declare a UDF taking arguments of SQL types *t1*, *t2*, and *t3*, returning type *t4*, it will be called as a Java method with the expected Java signature:

```
public void name (T1 a, T2 b, T3 c, T4 d)  { .....}
```

Where:

- *name* is the method name
- *T1* through *T4* are the Java types that correspond to SQL types *t1* through *t4.*
- *a*, *b*, and *c* are arbitrary variable names for the input arguments.
- *d* is an arbitrary variable name that represents the UDF result being computed.

For example, given a UDF called `sample!test3` that returns INTEGER and takes arguments of type CHAR(5), BLOB(10K), and DATE, DB2 expects the Java implementation of the UDF to have the following signature:

```
import COM.ibm.db2.app.*;
public class sample implements UDF {
   public void test3(String arg1, Blob arg2, String arg3,
                     int result) { ... }
}
```

Java UDFs that implement table functions have more arguments. Beside the variables representing the input, an additional variable appears for each column in the resulting row. For example, a table function may be declared as:

```
public void test4(String arg1,
             int result1, Blob result2, String result3);
```

SQL NULL values are represented by Java variables that are not initialized. These variables have a value of zero if they are primitive types, and Java null if they are object types, in accordance with Java rules. To tell an SQL NULL apart from an ordinary zero, you can call the function `isNull` for any input argument:

```
{ ....
  if (isNull(1)) { /* argument #1 was a SQL NULL */ }
  else           { /* not NULL */ }
}
```

In the above example, the argument numbers start at one. The `isNull()` function, like the other functions that follow, are inherited from the `COM.ibm.db2.app.UDF` interface. This must be implemented by Java classes containing UDFs.

To return a result from a scalar or table UDF, use the `set()` method in the UDF, as follows:

```
{ ....
  set(2, value);
}
```

Where '2' is the index of an output argument, and value is a literal or variable of a compatible type. The argument number is the index in the argument list of the selected output. In the first example in this section, the `int result` variable has an index of 4; in the second, `result1` through `result3` have indices of 2 through 4. An output argument that is not set before the UDF returns will have a NULL value.

Like C modules used in UDFs and stored procedures, you cannot use the Java standard I/O streams (`System.in`, `System.out`, and `System.err`) in Java UDFs. For an example of a Java UDF, see the file `DB2Udf.java` in the `sqllib/samples/java` directory.

Remember that all Java class files that you use to implement a UDF must reside in the `sqllib/function directory` or an appropriate subdirectory. See “Where to Put Java Classes” on page 514.

Changing How a Java UDF Runs

Typically, DB2 calls a UDF many times, once for each row of a result set in a query. The implementing Java class is instantiated once per row, and the selected method of each new instance is called once.

You can change this model by declaring the UDF with the SCRATCHPAD option. When you use this option, the Java class is instantiated only once, and the same instance is reused for the entire query. While C-language UDFs can maintain state between calls in a scratchpad area provided by the database engine, Java UDFs can simply use instance variables. Note that there is still a separate Java UDF instance per query reference to that UDF, just as there is for C UDFs. If a UDF is called in several places in a query, each call will have its own Java object.

At the end of a query, if you specify the FINAL CALL option on the CREATE FUNCTION statement, the object’s `public void close()` method is called. If you do not define this method, a stub function takes over and the event is ignored.

If you specify the ALLOW PARALLEL clause for a Java UDF in the CREATE FUNCTION statement, DB2 may elect to evaluate the UDF in parallel. If this occurs, several distinct Java objects may be created on different partitions. Each object receives a subset of the rows. Note that are no such object instances are created for C or C++ UDFs.

As with other UDFs, Java UDFs can be fenced or unfenced. Unfenced UDFs are run inside the address space of the database engine; fenced UDFs are run in a separate process. Although Java UDFs cannot inadvertently corrupt the address space of their embedding process, they can terminate or slow down the process. Therefore, when you are debugging UDFs written in Java, you should run them as fenced UDFs.

Refer to “COM.ibm.db2.app.UDF” on page 519 for a description of the `COM.ibm.db2.app.UDF` interface. This interface describes other useful calls that you can make within a UDF, such as `setSQLstate` and `getDBinfo`.

Creating and Using Java Stored Procedures

As with UDFs, you can create and use stored procedures in Java just like you can for other programming languages. There are some programming considerations (as discussed in “Coding Java Stored Procedures” on page 517) that you need to know when you write your Java code. You also need to *register* your Java stored procedure. Refer to the CREATE PROCEDURE statement in the *SQL Reference* for information on how to register your stored procedure.

Note: If you are running a *database server with local clients* node type, you must set the maxdari database manager configuration parameter to a non-zero value before you invoke a Java stored procedure.

A sample Java stored procedure, DB2Stp.java, is provided in sqllib/samples/java. For general details on DB2's stored procedure support, see Chapter 5, "Writing Stored Procedures" on page 167.

Remember that all Java class files that you use to implement a stored procedure must reside in the sqllib/function directory or appropriate subdirectory (as discussed in "Where to Put Java Classes" on page 514).

Coding Java Stored Procedures

Java stored procedures are public instance methods. Within the classes, the stored procedures are identified by their method name and signature. When you call a stored procedure, its signature is generated dynamically based on the variable types that you pass to it.

Java stored procedures are very similar to the Java UDFs described in "Creating and Using Java User-Defined Functions" on page 514. Like table functions, they can have multiple outputs. They also use the same conventions for NULL values, and the same set routine for output. The main difference is that a Java class that contains stored procedures must implement the COM.ibm.db2.app.StoredProc interface instead of the COM.ibm.db2.app.UDF interface. Refer to "COM.ibm.db2.app.StoredProc" on page 518 for a description of the COM.ibm.db2.app.StoredProc interface.

This interface provides the following routine to fetch a JDBC connection to the embedding application context:

```
public java.sql.Connection getConnection()
```

You can use this handle to run SQL statements. Other methods of the StoredProc interface are listed in the file sqllib/samples/java/StoredProc.java.

The following is a small stored procedure with one input and two outputs. It executes the given SQL query, and returns the number of rows in the result, and the SQLSTATE:

```
import COM.ibm.db2.app.*;
import java.sql.*;
public class sample2 implements StoredProc {
   public void donut(String query, int rowCount,
                     String sqlstate) throws Exception {
      try {
         Statement s = getConnection().createStatement();
         ResultSet r = s.executeQuery(query);
         int counter = 0;
         while(r.next()) {
             counter ++;
         }
         r.close();  s.close();
         set(2, counter);
      } catch(SQLException x) {
         set(3, x.getSQLState());
      }
   }
}
```

Classes for Java Stored Procedures and UDFs

There are four classes/interfaces that you can use with Java Stored Procedures or UDFs:

- COM.ibm.db2.app.StoredProc
- COM.ibm.db2.app.UDF
- COM.ibm.db2.app.Blob
- COM.ibm.db2.app.Clob

The following sections describe the public aspects of these classes' behavior:

COM.ibm.db2.app.StoredProc

A Java class that contains methods intended to be called as stored procedures must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-STP-class> implements COM.ibm.db2.app.StoredProc{ ... }
```

You can only call inherited methods of the `COM.ibm.db2.app.StoredProc` interface in the context of the currently executing stored procedure. For example, you cannot use operations on LOB arguments, result- or status-setting calls, etc., after a stored procedure returns. A Java exception will be thrown if you violate this rule.

Argument-related calls use a column index to identify the column being referenced. These start at 1 for the first argument. At this time, all arguments of a stored procedure are considered INOUT and thus are both inputs and outputs.

Any exception returned from the stored procedure is caught by the database and returned to the caller with SQLCODE -4302, SQLSTATE 38501. A JDBC SQLException or SQLWarning is handled specially and passes its own SQLCODE, SQLSTATE etc. to the calling application verbatim.

The following methods are associated with the `COM.ibm.db2.app.StoredProc` class:

```
public StoredProc() [default constructor]
```

This constructor is called by the database before the stored procedure call.

```
public boolean isNull(int) throws Exception
```

This function tests whether an input argument with the given index is an SQL NULL.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

This function sets the output argument with the given index to the given value. The index has to refer to a valid output argument, the data type must match, and the value must have an acceptable length and contents. Strings with Unicode characters must be representable in the database code page. Errors result in an exception being thrown.

```
public java.sql.Connection getConnection() throws Exception
```

This function returns a JDBC object that represents the calling application's connection to the database. It is analogous to the result of a null `SQLConnect()` call in a C stored procedure.

COM.ibm.db2.app.UDF

A Java class that contains methods intended to be called as UDFs must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-UDF-class> implements COM.ibm.db2.app.UDF{ ... }
```

You can only call methods of the `COM.ibm.db2.app.UDF` interface in the context of the currently executing UDF. For example, you cannot use operations on LOB arguments, result- or status-setting calls, etc., after a UDF returns. A Java exception will be thrown if this rule is violated.

Argument-related calls use a column index to identify the column being set. These start at 1 for the first argument. Output arguments are numbered higher than the input arguments. For example, a scalar UDF with three inputs uses index 4 for the output.

Any exception returned from the UDF is caught by the database and returned to the caller with SQLCODE -4302, SQLSTATE 38501.

The following methods are associated with the `COM.ibm.db2.app.UDF` class:

```
public UDF() [default constructor]
```

This constructor is called by the database at the beginning of a series of UDF calls. It precedes the first call to the UDF.

```
public void close()
```

This function is called by the database at the end of a UDF evaluation, if the UDF was created with the FINAL CALL option. It is analogous to the final call for a C UDF. If a Java UDF class does not implement this function, a no-op stub will handle and ignore this event.

```
public boolean isNull(int) throws Exception
```

This function tests whether an input argument with the given index is an SQL NULL.

```
public boolean needToSet(int) throws Exception
```

This function tests whether an output argument with the given index needs to be set. This may be false for a table UDF declared with DBINFO, if that column is not used by the UDF caller.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

This function sets the output argument with the given index to the given value. The index has to refer to a valid output argument, the data type must match, and the value must have an acceptable length and contents. Strings with Unicode characters must be representable in the database code page. Errors result in an exception being thrown.

```
public void setSQLstate(String) throws Exception
```

This function may be called from a UDF to set the SQLSTATE to be returned from this call. A table UDF should call this function with "02000" to signal the end-of-table condition. If the string is not acceptable as an SQLSTATE, an exception will be thrown.

```
public void setSQLmessage(String) throws Exception
```

This function is similar to the `setSQLstate` function. It sets the SQL message result. If the string is not acceptable (for example, longer than 70 characters), an exception will be thrown.

```
public String getFunctionName() throws Exception
```

This function returns the name of the executing UDF.

```
public String getSpecificName() throws Exception
```

This function returns the specific name of the executing UDF.

```
public byte[] getDBinfo() throws Exception
```

This function returns a raw, unprocessed DBINFO structure for the executing UDF, as a byte array. You must first declare it with the DBINFO option.

```
public String getDBname() throws Exception
public String getDBauthid() throws Exception
public String getDBtbschema() throws Exception
public String getDBtbname() throws Exception
public String getDBcolname() throws Exception
public String getDBver_rel() throws Exception
public String getDBplatform() throws Exception
```

These functions return the value of the appropriate field from the DBINFO structure of the executing UDF.

```
public int[] getDBcodepg() throws Exception
```

This function returns the SBCS, DBCS, and composite code page numbers for the database, from the DBINFO structure. The returned integer array has the respective numbers as its first three elements.

```
public byte[] getScratchpad() throws Exception
```

This function returns a copy of the scratchpad of the currently executing UDF. You must first declare the UDF with the SCRATCHPAD option.

```
public void setScratchpad(byte[]) throws Exception
```

This function overwrites the scratchpad of the currently executing UDF with the contents of the given byte array. You must first declare the UDF with the SCRATCHPAD option. The byte array must have the same size as `getScratchpad()` returns.

COM.ibm.db2.app.Blob

An instance of this class is passed by the database to represent a BLOB as UDF or stored procedure input, and may be passed back as output. The application may create instances, but only in the context of an executing UDF or stored procedure. Uses of these objects outside such a context will throw an exception.

The following methods are associated with the `COM.ibm.db2.app.Blob` class:

```
public static COM.ibm.db2.app.Blob new() throws Exception
```

This function creates a temporary Blob. It will be implemented using a LOCATOR if possible.

```
public long size() throws Exception
```

This function returns the length (in bytes) of the BLOB.

```
public java.io.InputStream getInputStream() throws Exception
```

This function returns a new InputStream to read the contents of the BLOB. Efficient seek/mark operations are available on that object.

```
public java.io.OutputStream getOutputStream() throws Exception
```

This function returns a new OutputStream to append bytes to the BLOB. Appended bytes become immediately visible on all existing InputStream instances produced by this object's `getInputStream()` call.

COM.ibm.db2.app.Clob

An instance of this class is passed by the database to represent a CLOB or DBCLOB as UDF or stored procedure input, and may be passed back as output. The application may create instances, but only in the context of an executing UDF or stored procedure. Uses of these objects outside such a context will throw an exception.

Clob instances store characters in the database code page. Some Unicode characters may not be representable in this code page, and may cause an exception to be thrown during conversion. This may happen during an append operation, or during a UDF or StoredProc `set()` call. This is necessary to hide the distinction between a CLOB and a DBCLOB from the Java programmer.

The following methods are associated with the `COM.ibm.db2.app.Clob` class:

```
public static COM.ibm.db2.app.Clob new() throws Exception
```

This function creates a temporary Clob. It will be implemented using a LOCATOR if possible.

```
public long size() throws Exception
```

This function returns the length (in characters) of the CLOB.

```
public java.io.Reader getReader() throws Exception
```

This function returns a new Reader to read the contents of the CLOB or DBCLOB. Efficient seek/mark operations are available on that object.

```
public java.io.Writer getWriter() throws Exception
```

This function returns a new Writer to append characters to this CLOB or DBCLOB. Appended characters become immediately visible on all existing Reader instances produced by this object's `GetReader()` call.

Appendix A. Supported SQL Statements (DB2 Universal Database)

Table 22:

- Lists all the supported SQL statements
- Indicates (with an 'X') if they can be executed dynamically
- Indicates (with an 'X') if they are supported by the command line processor (CLP)
- Indicates (with an 'X' or CLI function name) if the statement can be executed using the call level interface (CLI).

You can use Table 22 as a quick reference aid. For a complete discussion of all the statements, including their syntax, see the *SQL Reference*.

Table 22 (Page 1 of 3). SQL Statements (DB2 Universal Database)

SQL Statement	**Dynamic**[1]	**Command Line Processor (CLP)**	**Call Level Interface**[3] **(CLI)**
ALTER { BUFFERPOOL, NODEGROUP, TABLE, TABLESPACE }	X	X	X
BEGIN DECLARE SECTION[2]			
CALL			X[4]
CLOSE		X	SQLCloseCursor(), SQLFreeStmt()
COMMENT ON	X	X	X
COMMIT	X	X	SQLEndTran, SQLTransact()
Compound SQL			X[4]
CONNECT (Type 1)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()
CONNECT (Type 2)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, INDEX, NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRIGGER, VIEW }	X	X	X
DECLARE CURSOR[2]		X	SQLAllocStmt()
DELETE	X	X	X
DESCRIBE[8]		X	SQLColAttributes(), SQLDescribeCol(), SQLDescribParam()[6]
DISCONNECT		X	SQLDisconnect()
DROP	X	X	X
END DECLARE SECTION[2]			
EXECUTE			SQLExecute()

Table 22 (Page 2 of 3). SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)
EXECUTE IMMEDIATE			`SQLExecDirect()`
EXPLAIN	X	X	X
FETCH		X	`SQLExtendedFetch()`[7] , `SQLFetch()`, `SQLFetchScroll()`[7]
FREE LOCATOR			X[4]
GRANT	X	X	X
INCLUDE[2]			
INSERT	X	X	X
LOCK TABLE	X	X	X
OPEN		X	`SQLExecute()`, `SQLExecDirect()`
PREPARE			`SQLPrepare()`
RELEASE		X	
RENAME TABLE	X	X	X
REVOKE	X	X	X
ROLLBACK	X	X	`SQLEndTran()`, `SQLTransact()`
select-statement	X	X	X
SELECT INTO			
SET CONNECTION		X	`SQLSetConnection()`
SET CONSTRAINTS	X	X	X
SET CURRENT DEGREE	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, `SQLSetConnectAttr()`
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, `SQLSetConnectAttr()`
SET CURRENT FUNCTION PATH	X	X	X
SET CURRENT PACKAGESET			
SET CURRENT QUERY OPTIMIZATION	X	X	X
SET EVENT MONITOR STATE	X	X	X
SET transition-variable[5]	X	X	X
SIGNAL SQLSTATE[5]	X	X	X
UPDATE	X	X	X
VALUES INTO			
WHENEVER[2]			

Table 22 (Page 3 of 3). SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)

Notes:

1. You can code all statements in this list as static SQL, but only those marked with X as dynamic SQL.
2. You cannot execute this statement.
3. An X indicates that you can execute this statement using either `SQLExecDirect()` or `SQLPrepare()` and `SQLExecute()`. If there is an equivalent DB2 CLI function, the function name is listed.
4. Although this statement is not dynamic, with DB2 CLI you can specify this statement when calling either `SQLExecDirect()`, or `SQLPrepare()` and `SQLExecute()`.
5. You can only use this within CREATE TRIGGER statements.
6. You can only use the SQL DESCRIBE statement to describe output, whereas with DB2 CLI you can also describe input (using the `SQLDescribeParam()` function).
7. You can only use the SQL FETCH statement to fetch one row at a time in one direction, whereas with the DB2 CLI `SQLExtendedFetch()` and `SQLFetchScroll()` functions, you can fetch into arrays. Furthermore, you can fetch in any direction, and at any position in the result set.
8. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command. For information on the DESCRIBE SQL statement, refer to the *SQL Reference*. For information on the DESCRIBE CLP command, refer to the *Command Reference*.

Appendix B. Sample Programs and Extra Examples

This section provides information on the programming examples supplied with DB2. Some examples that are not described in the main sections of this book are listed here.

All sample programs can be found in the `samples` subdirectory of the `sqllib` directory. There is a subdirectory for each supported language; however, all sample programs may not be available in all supported languages. Table 24 on page 531 shows a list of the sample programs that are available. Refer to the *API Reference* for a list of the available API sample code.

Installing, Building and Executing the Sample Programs

The sample programs used in this book show examples of embedded SQL statements and API calls in the supported host languages. The sample programs are written to be short and simple. Production applications should check the return codes, and especially the SQLCODE or SQLSTATE from all API calls and SQL statements. For information on handling error conditions, SQLCODEs, and SQLSTATEs, see “Diagnostic Handling and the SQLCA Structure” on page 83. See the *DB2 SDK Building Applications* for details on how to install, build, and execute these programs in your environment.

The DB2 SDK comes with sample programs. The file extensions for each supported language, and the directories where the programs can be found on the supported platforms, are given in Table 23 on page 528.

The sample programs providing examples of embedded SQL and DB2 API calls are shown in Table 24 on page 531. Command Line Processor (CLP) files provided by DB2 are shown in Table 25 on page 537.

Java sample programs are shown in Table 26 on page 537. Object Linking and Embedding (OLE) sample programs are shown in Table 27 on page 538. The sample programs demonstrating DB2 CLI calls are shown in Table 28 on page 538.

You can use the sample programs to learn how to code your applications.

Table 23. Sample Program File Extensions and Locations

Language		**CLI Programs**	**Programs with Embedded SQL**	**Programs without Embedded SQL**
C	File Ext.	`.c`	`.sqc`	`.c`
	Directory	`samples/cli`	`samples/c`	`samples/c`
C++	File Ext.	`Not Applicable`	`.sqC` (UNIX) `.sqx`	`.C` (UNIX) `.cxx` (Intel)
	Directory	`Not Applicable`	`samples/cpp`	`samples/cpp`
COBOL	File Ext.	`Not Applicable`	`.sqb`	`.cbl`
	Directory	`Not Applicable`	`samples/cobol` `samples/cobol_mf`	`samples/cobol` `samples/cobol_mf`
FORTRAN	File Ext.	`Not Applicable`	`.sqf`	`.f` (UNIX) `.for` (OS/2)
	Directory	`Not Applicable`	`samples/fortran`	`samples/fortran`
REXX	File Ext.	`Not Applicable`	`.cmd`	`.cmd`
	Directory	`Not Applicable`	`samples/rexx`	`samples/rexx`
JAVA	File Ext.	`Not Applicable`	`Not Applicable`	`.java`
	Directory	`Not Applicable`	`Not Applicable`	`samples/java`
OLE	File Ext.	`Not Applicable`	`Not Applicable`	`Not Applicable`
	Directory	`samples\ole`	`Not Applicable`	`samples\ole`

Note:

Programs without SQL Denotes programs with no SQL statements in them (primarily programs using DB2 API functions).

Directory Delimiters On UNIX are `/`. On OS/2 and Windows platforms, are `\`.

IBM COBOL samples Are only supplied on the OS/2, AIX, Windows NT and Windows 95 platforms in the `cobol` subdirectory.

Micro Focus Cobol Samples Are supplied on all platforms except the Macintosh. The 16-bit Micro Focus COBOL examples are supplied in the `cobol_16` subdirectory on OS/2, and the `cobol` subdirectory on Windows 3.1. For all other platforms, the Micro Focus COBOL samples are in the `cobol_mf` subdirectory.

Fortran Samples Are only supplied on the AIX, HP-UX, Silicon Graphics IRIX, Solaris, and OS/2 platforms.

REXX Samples Are only supplied on the AIX, OS/2, Windows NT and Windows 95 platforms.

Java Samples Are stored procedures and UDFs, as well as Java Database Connectivity (JDBC) applications and applets. Java samples are available on the AIX, HP-UX, Solaris, OS/2, Windows NT and Windows 95 platforms.

OLE Samples Are for Object Linking and Embedding (OLE) in Microsoft Visual Basic and Microsoft Visual C++, supplied on the Windows NT and Windows 95 platforms only.

The above table lists the supported languages within the specified programming paradigms. Not all sample programs have been ported to all the supported programming languages.

You can find the sample programs in the `samples` subdirectory of the directory where DB2 has been installed. There is a subdirectory for each supported language. The following examples show you how to locate the samples written in C or C++ on each supported platform.

- On UNIX platforms.

  You can find the C source code for embedded SQL and DB2 API programs in `sqllib/samples/c` under your database instance directory; the C source code for DB2 CLI programs is in `sqllib/samples/cli`. For additional information about the sample programs in Table 24 on page 531 and Table 28 on page 538, refer to the `README` file in the appropriate `samples` subdirectory under your database manager instance. The `README` file will contain any additional samples that are not listed in this book.

- On OS/2, Windows NT, and Windows 95 platforms.

  You can find the C source code for embedded SQL and DB2 API programs in `%DB2PATH%\samples\c` under the DB2 install directory; the C source code for DB2 CLI programs is in `%DB2PATH%\samples\cli`. The variable `%DB2PATH%` determines where DB2 is installed. Depending on which drive DB2 is installed, `%DB2PATH%` will point to *drive*:`\sqllib`. For additional information about the sample programs in Table 24 on page 531 and Table 28 on page 538, refer to the `README` file in the appropriate `%DB2PATH%\samples` subdirectory. The `README` file will contain any additional samples that are not listed in this book.

- On Windows 3.1.

  You can find the C source code for embedded SQL and DB2 API programs in `%DB2PATH%\samples\c`; the C source code for DB2 CLI programs is in `%DB2PATH%\samples\cli`. The `db2.ini` file, which stores the DB2 settings, defines the value for `%DB2PATH%`, which by default points to *drive*`:\sqllib\win`. The value of `%DB2PATH%`, as referenced in the `db2.ini` file, is only recognized within the DB2 environment. For additional information about the sample programs in Table 24 on page 531 and Table 28 on page 538, refer to the `README` files in these subdirectories. The `README` files will contain any additional samples that are not listed in this book.

- On Macintosh.

  You can find the sample programs in the `DB2:samples:` folder. There are sub-folders for sample programs written in C and CLI. For additional information about the sample programs in Table 24 on page 531 and Table 28 on page 538, refer to the README file in the `DB2:samples:` folder. The README file will contain any additional samples that are not listed in this book.

If your platform is not addressed in Table 23 on page 528, please refer to the *DB2 SDK Building Applications* book for your platform for information specific to your environment.

Not all of the sample programs are available in all the supported programming languages.

The sample programs directory is typically read-only on most platforms. Before you alter or build the sample programs, copy them to your working directory. On the Macintosh, copy them to your working folder.

Note: The sample programs that are shipped with DB2 Universal Database have dependencies on the English version of the Sample database and the associated table and column names. If the Sample database has been translated into another national language on your version of DB2 Universal Database, you need to update the name of the Sample database, and the names of the tables and the columns coded in the supplied sample programs, to the names used in the translated Sample database. Otherwise, you will experience problems running the sample programs as shipped.

Currently, the Sample database is translated for the following countries:

- France
- Italy
- Spain
- Finland
- Norway
- People’s Republic of China

In Table 24 on page 531, ‘Yes’, in the *Embedded SQL* column, indicates that the program contains embedded SQL. A blank indicates that the program does not contain embedded SQL, and thus no precompiling is required.

Table 24 (Page 1 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
`adhoc`	Yes	Demonstrates dynamic SQL and the SQLDA structure to process SQL commands interactively. SQL commands are input by the user, and output corresponding to the SQL command is returned. See "Example ADHOC Program" on page 113 for details.
`advsql`	Yes	Demonstrates the use of advanced SQL expressions like CASE, CAST, and scalar full selects.
`asynrlog`	Yes	Demonstrates the use of the following API: `ASYNCHRONOUS LOG READ`
`autoloader`		A UNIX Korn shell script that prepares ftp scripts for data transfer from remote hosts and generates a temporary buffer space (FIFO or named pipes). It then starts `db2split` and invokes DB2 LOAD. In a partitioned environment, partitioning keys are used to determine the partition where the data resides. Therefore, data must pass through a *splitting* phase before it can be loaded at the correct partition. The entire *split and load* process can be accomplished by the `autoLoader` utility. It uses a system-defined hashing function to partition the data into as many output files as there are partitions in the nodegroup in which the table is defined. It then loads these output files concurrently across the set of partitions in the nodegroup.

`backrest`		Demonstrates the use of the following APIs:
		`BACKUP DATABASE`
		`RESTORE DATABASE`
		`ROLL FORWARD DATABASE`

`blobfile`	Yes	Demonstrates the manipulation of a Binary Large Object (BLOB), by reading a BLOB value from the sample database and placing it in a file, the contents of which can be displayed using an external viewer.
`bindfile`	Yes	Demonstrates the use of the BIND API to bind an embedded SQL application to a database.
`calludf`	Yes	Demonstrates the use of the library of User-Defined Functions (UDFs) created by `udf` for the SAMPLE database tables.

`client`		Demonstrates the use of the following APIs:
		`SET CLIENT`
		`QUERY CLIENT`

`columns`	Yes	Demonstrates the use of a cursor that is processed using dynamic SQL. This program lists all the entries in the system table, SYSIBM.SYSTABLES, under a desired schema name.
`cursor`	Yes	Demonstrates the use of a cursor using static SQL. See "Example Cursor Program" on page 58 for details.
`d_dbconf`		Demonstrates the use of the following API: `GET DATABASE CONFIGURATION DEFAULTS`
`d_dbmcon`		Demonstrates the use of the following API: `GET DATABASE MANAGER CONFIGURATION DEFAULTS`

Table 24 (Page 2 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
da_manip	Yes	Provides a library of routines to manipulate SQLDAs and SQLVARs.
db2mon		Demonstrates how to use the Database System Monitor APIs, and how to process the output data buffer returned from the Snapshot API.
db2uext2		Provides a sample log management user exit.
dbauth	Yes	Demonstrates the use of the following API: GET AUTHORIZATIONS
dbcat		Demonstrates the use of the following APIs: CATALOG DATABASE CLOSE DATABASE DIRECTORY SCAN GET NEXT DATABASE DIRECTORY ENTRY OPEN DATABASE DIRECTORY SCAN UNCATALOG DATABASE
dbcmt		Demonstrates the use of the following APIs: CHANGE DATABASE COMMENT
dbconf		Demonstrates the use of the following APIs: CREATE DATABASE DROP DATABASE GET DATABASE CONFIGURATION RESET DATABASE CONFIGURATION UPDATE DATABASE CONFIGURATION
dbinst		Demonstrates the use of the following APIs: ATTACH TO INSTANCE DETACH FROM INSTANCE GET INSTANCE
dbmconf		Demonstrates the use of the following APIs: GET DATABASE MANAGER CONFIGURATION RESET DATABASE MANAGER CONFIGURATION UPDATE DATABASE MANAGER CONFIGURATION
dbsnap		Demonstrates the use of the following API: DATABASE SYSTEM MONITOR SNAPSHOT
dbstart		Demonstrates the use of the following API: START DATABASE MANAGER
dbstat	Yes	Demonstrates the use of the following APIs: REORGANIZE TABLE RUN STATISTICS
dbstop		Demonstrates the use of the following APIs: FORCE USERS STOP DATABASE MANAGER

Table 24 (Page 3 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
db_udcs		Demonstrates the use of the following APIs in order to simulate the collating behaviour of a DB2 for MVS/ESA or OS/390 CCSID 500 (EBCDIC International) collating sequence: CREATE DATABASE DROP DATABASE
dcscat		Demonstrates the use of the following APIs: ADD DCS DIRECTORY ENTRY CLOSE DCS DIRECTORY SCAN GET DCS DIRECTORY ENTRY FOR DATABASE GET DCS DIRECTORY ENTRIES OPEN DCS DIRECTORY SCAN UNCATALOG DCS DIRECTORY ENTRY
delet	Yes	Demonstrates static SQL to delete items from a database.
dmscont		Demonstrates the use of the following APIs in order to create a database with more than one database managed storage (DMS) container: CREATE DATABASE DROP DATABASE
dynamic	Yes	Demonstrates the use of a cursor using dynamic SQL. See “Example Dynamic SQL Program” on page 96 for details.
ebcdicdb		Demonstrates the use of the following APIs in order to simulate the collating behaviour of a DB2 for MVS/ESA or OS/390 CCSID 037 (EBCDIC US English) collating sequence: CREATE DATABASE DROP DATABASE
expsamp	Yes	Demonstrates the use of the following APIs: EXPORT IMPORT in conjunction with a DRDA database.
fillcli	Yes	Demonstrates the client-side of a stored procedure that uses the SQLDA to pass information specifying which table the stored procedure populates with random data.
fillsrv	Yes	Demonstrates the server-side of a stored procedure example that uses the SQLDA to receive information from the client specifying the table that the stored procedure populates with random data.
impexp	Yes	Demonstrates the use of the following APIs: EXPORT IMPORT

Table 24 (Page 4 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
inpcli	Yes	Demonstrates stored procedures using either the SQLDA structure or host variables. This is the client program of a client/server example. (The server program is called inpsrv.) The program fills the SQLDA with information, and passes it to the server program for further processing. The SQLCA status is returned to the client program. This program shows the invocation of stored procedures using an embedded SQL CALL statement. See "How the Example Input-SQLDA Client Application Works" on page 203 for details.
inpsrv	Yes	Demonstrates stored procedures using the SQLDA structure. This is the server program of a client/server example. (The client program is called inpcli.) The program creates a table (PRESIDENTS) in the SAMPLE database with the information received in the SQLDA. The server program does all the database processing and returns the SQLCA status to the client program. See "How the Example Input-SQLDA Stored Procedure Works" on page 215 for details.
joinsql	Yes	An example using advanced SQL join expressions.
largevol	Yes	Demonstrates parallel query processing in a partitioned environment, and the use of an NFS file system to automate the merging of the result sets. See "Example: Extracting Large Volume of Data (largevol.c)" on page 410 for details.
lobeval	Yes	Demonstrates the use of LOB locators and deferring the evaluation of the actual LOB data. See "How the Sample LOBEVAL Program Works" on page 243 for details.
lobfile	Yes	Demonstrates the use of LOB file handles. See "How the Sample LOBFILE Program Works" on page 250 for details.
lobloc	Yes	Demonstrates the use of LOB locators. See "How the Sample LOBLOC Program Works" on page 234 for details.
loblocud		Demonstrates the use of LOB locators in a user-defined function. See "Example: Function using LOB locators" on page 338 for details.
lobval	Yes	Demonstrates the use of LOBs.
makeapi	Yes	Demonstrates the use of the following APIs: BIND PRECOMPILE PROGRAM START DATABASE MANAGER STOP DATABASE MANAGER
migrate		Demonstrates the use of the following API: MIGRATE DATABASE
monreset		Demonstrates the use of the following API: RESET DATABASE SYSTEM MONITOR DATA AREAS
monsz		Demonstrates the use of the following APIs: ESTIMATE DATABASE SYSTEM MONITOR BUFFER SIZE DATABASE SYSTEM MONITOR SNAPSHOT

Table 24 (Page 5 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
nodecat		Demonstrates the use of the following APIs: CATALOG NODE CLOSE NODE DIRECTORY SCAN GET NEXT NODE DIRECTORY ENTRY OPEN NODE DIRECTORY SCAN UNCATALOG NODE
openftch	Yes	Demonstrates fetching, updating, and deleting of rows using static SQL. See "How the Example OPENFTCH SQL Program Works" on page 65 for details.
outcli	Yes	Demonstrates stored procedures using the SQLDA structure. This is the client program of a client/server example. (The server program is called outsrv.) This program allocates and initializes a one variable SQLDA, and passes it to the server program for further processing. The filled SQLDA is returned to the client program along with the SQLCA status. This program shows the invocation of stored procedures using an embedded SQL CALL statement. See "How the Example Output-SQLDA Client Application Works" on page 183 for details.
outsrv	Yes	Demonstrates stored procedures using the SQLDA structure. This is the server program of a client/server example. (The client program is called outcli.) The program fills the SQLDA with the median SALARY of the employees in the STAFF table of the SAMPLE database. The server program does all the database processing (finding the median). The server program returns the filled SQLDA and the SQLCA status to the client program. See "How the Example Output-SQLDA Stored Procedure Works" on page 193 for details.
qload	Yes	Demonstrates the use of the following API: LOAD QUERY
rebind	Yes	Demonstrates the use of the following API: REBIND PACKAGE
rechist		Demonstrates the use of the following APIs: CLOSE RECOVERY HISTORY FILE SCAN GET NEXT RECOVERY HISTORY FILE ENTRY OPEN RECOVERY HISTORY FILE SCAN PRUNE RECOVERY HISTORY FILE ENTRY UPDATE RECOVERY HISTORY FILE ENTRY
recursql	Yes	Demonstrates the use of advanced SQL recursive queries.
regder		Demonstrates the use of the following APIs: REGISTER DEREGISTER
restart		Demonstrates the use of the following API: RESTART DATABASE
sampudf	Yes	Demonstrates the use of User-Defined Types (UDTs) and User-Defined Functions (UDFs). The UDFs declared in this program are all sourced UDFs.

Table 24 (Page 6 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
`setact`		Demonstrates the use of the following API: `SET ACCOUNTING STRING`
`setrundg`		Demonstrates the use of the following API: `SET RUNTIME DEGREE`
`static`	Yes	Uses static SQL to retrieve information. See “Example Static SQL Program” on page 41 for details.
`sws`		Demonstrates the use of the following API: `DATABASE MONITOR SWITCH`
`system`		Demonstrates most of the system-specific calls.
`tabinfo`	Yes	Provides a library of routines for obtaining table and column information from the system tables and for accessing the information obtained.

`tabscont`		Demonstrates the use of the following APIs:
		`TABLESPACE CONTAINER QUERY`
		`OPEN TABLESPACE CONTAINER QUERY`
		`FETCH TABLESPACE CONTAINER QUERY`
		`CLOSE TABLESPACE CONTAINER QUERY`
		`SET TABLESPACE CONTAINER QUERY`

`tabspace`		Demonstrates the use of the following APIs:
		`TABLESPACE QUERY`
		`SINGLE TABLESPACE QUERY`
		`OPEN TABLESPACE QUERY`
		`FETCH TABLESPACE QUERY`
		`GET TABLESPACE STATISTICS`
		`CLOSE TABLESPACE QUERY`

`tabsql`	Yes	Demonstrates the use of advanced SQL table expressions.
`tblcli`		Demonstrates a call to a table function (client-side) to display weather information for a number of cities.
`tblsrv`		Demonstrates a table function (server-side) that processes weather information for a number of cities.

`tload`	Yes	Demonstrates the use of the following APIs:
		`EXPORT`
		`QUIESCE TABLESPACE FOR TABLES`
		`LOAD`

`trigsql`	Yes	An example using advanced SQL triggers and constraints.
`udf`	Yes	Creates a library of User-Defined Functions (UDFs) made specifically for the SAMPLE database tables, but can be used with tables of compatible column types.
`updat`	Yes	Uses static SQL to update a database. See “Example UPDAT Program” on page 75 for details.

Table 24 (Page 7 of 7). Sample Programs Showing Embedded SQL and APIs

Sample Program Name	Embedded SQL	Program Description
util		Demonstrates the use of the following APIs: GET ERROR MESSAGE GET SQLSTATE MESSAGE INSTALL SIGNAL HANDLER INTERRUPT This program also contains code to output information from an SQLDA. See “Using GET ERROR MESSAGE in Example Programs” on page 85 for details.
varinp	Yes	An example of variable input to Embedded Dynamic SQL statement calls using parameter markers. See “How the Example VARINP SQL Program Works” on page 120 for details.

Table 25. Command Line Processor (CLP) Sample Files.

Sample File Name	File Description
const.clp	Creates a table with a CHECK CONSTRAINT clause.
cte.clp	Demonstrates a common table expression. The equivalent sample program demonstrating this advanced SQL statement is tabsql.
flt.clp	Demonstrates a recursive query. The equivalent sample program demonstrating this advanced SQL statement is recursql.
join.clp	Demonstrates an outer join of tables. The equivalent sample program demonstrating this advanced SQL statement is joinsql.
stock.clp	Demonstrates the use of triggers. The equivalent sample program demonstrating this advanced SQL statement is trigsql.
testdata.clp	Uses DB2 built-in functions such as RAND() and TRANSLATE() to populate a table with randomly generated test data.

Table 26 (Page 1 of 2). Java Sample Programs

Sample Program Name	Program Description
DB2Appl.java	A Java Database Connectivity (JDBC) application that queries the sample database using the invoking user's privileges.
DB2Applt.java	A Java Database Connectivity (JDBC) applet that queries the sample database using a user and server specified as applet parameters.
DB2Applt.html	An HTML file that embeds the DB2Applt.java applet sample program. It needs to be customized with server and user information.
DB2Stp.java	A Java stored procedure that updates the EMPLOYEE table on the server, and returns new salary and payroll information to the client.

Table 26 (Page 2 of 2). Java Sample Programs

Sample Program Name	**Program Description**
DB2Udf.java	A Java user-defined function (UDF) that demonstrates several tasks, including integer division, manipulation of Character Large OBjects (CLOBs), and the use of Java instance variables.
samples.zip	A file containing compiled .class files for all DB2 Java samples.

Table 27. Object Linking and Embedding (OLE) Sample Programs

Sample Program Name	**Program Description**
sales	Demonstrates rollup queries on a Microsoft Excel sales spreadsheet (implemented in Visual Basic).
names	Queries a Lotus Notes address book (implemented in Visual Basic).
inbox	Queries Microsoft Exchange inbox e-mail messages through OLE/Messaging (implemented in Visual Basic).
invoice	An OLE automation user-defined function that sends Microsoft Word invoice documents as e-mail attachments (implemented in Visual Basic).
ccounter	A counter OLE automation user-defined function (implemented in Visual C++).
salarysrv	An OLE automation stored procedure that calculates the median salary of the STAFF table of the SAMPLE database (implemented in Visual Basic).
salaryclt	A client program that invokes the median salary OLE automation stored procedure salarysrv (implemented in Visual Basic and in Visual C++).

Table 28 (Page 1 of 3). Sample CLI Programs in DB2 Universal Database

Sample Program Name	**Program Description**
Utility files used by most CLI samples	
samputil.c	Utility functions used by most samples
samputil.h	Header file for samputil.c, included by most samples
General CLI Samples	
adhoc.c	Interactive SQL with formatted output (was typical.c)
async.c **	Run a function asynchronously (based on fetch.c)
basiccon.c	Basic connection
browser.c	List columns, foreign keys, index columns or stats for a table
colpriv.c	List column Privileges
columns.c	List all columns for table search string
compnd.c	Compound SQL example
datasour.c	List all available data sources

Table 28 (Page 2 of 3). Sample CLI Programs in DB2 Universal Database

Sample Program Name	Program Description
descrptr.c **	Example of descriptor usage
drivrcon.c	Rewrite of basiccon.c using SQLDriverConnect
duowcon.c	Multiple DUOW Connect type 2, syncpoint 1 (one phase commit)
embedded.c	Show equivalent DB2 CLI calls, for embedded SQL (in comments)
fetch.c	Simple example of a fetch sequence
getattrs.c	List some common environment, connection and statement options/attributes
getcurs.c	Show use of SQLGetCursor, and positioned update
getdata.c	Rewrite of fetch.c using SQLGetData instead of SQLBindCol
getfuncs.c	List all supported functions
getfuncs.h	Header file for getfuncs.c
getinfo.c	Use SQLGetInfo to get driver version and other information
getsqlca.c	Rewrite of adhoc.c to use prepare/execute and show cost estimate
lookres.c	Extract string from resume clob using locators
mixed.sqc	CLI sample with functions written using embedded SQL (Note: This file must be precompiled)
multicon.c	Multiple connections
native.c	Simple example of calling SQLNativeSql, and SQLNumParams
prepare.c	Rewrite of fetch.c, using prepare/execute instead of execdirect
proccols.c	List procedure parameters using SQLProcedureColumns
procs.c	List procedures using SQLProcedures
sfetch.c **	Scrollable cursor example (based on xfetch.c)
setcolat.c	Set column attributes (using SQLSetColAttributes)
setcurs.c	Rewrite of getcurs.c using SQLSetCurs for positioned update
seteattr.c	Set environment attribute (SQL_ATTR_OUTPUT_NTS)
tables.c	List all tables
typeinfo.c	Display type information for all types for current data source
xfetch.c	Extended Fetch, multiple rows per fetch
BLOB Samples	
picin.c	Loads graphic BLOBS into the emp_photo table directly from a file using SQLBindParamToFile
picin2.c	Loads graphic BLOBS into the emp_photo table using SQLPutData
showpic.c	Extracts BLOB picture to file (using SQLBindColToFile), then displays the graphic.
showpic2.c	Extracts BLOB picture to file using piecewise output, then displays the graphic.
Stored Procedure Samples	
clicall.c	Defines a CLI function which is used in the embedded SQL sample mrspcli3.sqc

Table 28 (Page 3 of 3). Sample CLI Programs in DB2 Universal Database

Sample Program Name	**Program Description**
inpcli.c	Call embedded input stored procedure samples/c/inpsrv
inpcli2.c	Call CLI input stored procedure inpsrv2
inpsrv2.c	CLI input stored procedure (rewrite of embedded sample inpsrv.sqc)
mrspcli.c	CLI program that calls mrspsrv.c
mrspcli2.c	CLI program that calls mrspsrv2.sqc
mrspcli3.sqc	An embedded SQL program that calls mrspsrv2.sqc using clicall.c
mrspsrv.c	Stored procedure that returns a multi-row result set
mrspsrv2.sqc	An embedded SQL stored procedure that returns a multi-row result set
outcli.c	Call embedded output stored procedure samples/c/inpsrv
outcli2.c	Call CLI output stored procedure inpsrv2
outsrv2.c	CLI output stored procedure (rewrite of embedded sample inpsrv.sqc)
Samples using ORDER tables created by create.c (Run in the following order)	
create.c	Creates all tables for the order scenario
custin.c	Inserts customers into the customer table (array insert)
prodin.c	Inserts products into the products table (array insert)
prodpart.c	Inserts parts into the prod_parts table (array insert)
ordin.c	Inserts orders into the ord_line, ord_cust tables (array insert)
ordrep.c	Generates order report using multiple result sets
partrep.c	Generates exploding parts report (recursive SQL Query)
order.c	UDF library code (declares a 'price' UDF)
order.exp	Used to build order libary
Version 2 Samples unchanged	
v2sutil.c	samputil.c using old v2 functions
v2sutil.h	samputil.h using old v2 functions
v2fetch.c	fetch.c using old v2 functions
v2xfetch.c	xfetch.c using old v2 functions

Note: Samples marked with a ** are new for this release.

Other files in the `samples/cli` directory include:

- README - Lists all example files.
- makefile - Makefile for all files

Appendix C. Programming in a DRDA Environment

This section contains information that is common to the *DB2 Connect User's Guide*. If you encounter an unfamiliar term or concept in this section, see the *DB2 Connect User's Guide*.

DB2 Connect lets an application program access data in any DRDA server database. For example, an application running on OS/2 can access data in a DB2 for OS/390 database. You can create new applications, or modify existing applications to run in a DRDA environment. You can also develop applications in one environment and port them to another.

DB2 Connect enables you to use the following items with host database products such as DB2 for OS/390, as long as the item is supported by the host database product:

- Embedded SQL, both static and dynamic
- Compound SQL, as described in “NOT ATOMIC Compound SQL” on page 550
- Stored procedures, as described in “Stored Procedures” on page 549
- The DB2 Call Level Interface
- The Microsoft ODBC API
- JDBC.

Some SQL statements differ among relational database products. You may encounter the following situations:

- SQL statements that are the same for all the database products that you use regardless of standards
- SQL statements that are documented in the *SQL Reference* and are therefore available in all IBM relational database products
- SQL statements that are unique to one database system that you access.

SQL statements in the first two categories are highly portable, but those in the third category will first require changes.

In general, SQL statements in Data Definition Language (DDL) are not as portable as those in Data Manipulation Language (DML).

DB2 Connect accepts some SQL statements that are not supported by DB2 Universal Database. DB2 Connect passes these statements on to the DRDA server.

For information on limits on different platforms, such as the maximum column length, see the sections on *SQL Limits* and *Considerations for Using* in the *SQL Reference*.

If you move a CICS application from OS/390 or VSE to run under another CICS product (for example, CICS for AIX), it can also access the OS/390 or VSE database using DB2 Connect. Refer to the *CICS/6000 Application Programming Guide* and the *CICS Customization and Operation* manual for more details.

Note: You can use DB2 Connect with a DB2 Universal Database Version 5 database using DRDA, although it would be more efficient to use the DB2 private protocol without DB2 Connect. Most of the incompatibility issues listed in the following sections will not apply if you are using DB2 Connect against a DB2 Universal Database Version 5 database, except in cases where a restriction is due to a limitation of DRDA itself, for example, the non-support of large object (LOB) data types.

When you program in a DRDA environment, you should consider the following specific factors:

- Using Data Definition Language (DDL)
- Using Data Manipulation Language (DML)
- Using Data Control Language (DCL)
- Connecting and disconnecting
- Precompiling
- Defining a sort order
- Managing referential integrity
- Locking
- Differences in SQLCODEs and SQLSTATEs
- Using system catalogs
- Isolation levels
- Stored procedures
- NOT ATOMIC compound SQL
- Distributed unit of work
- SQL statements supported or rejected by DB2 Connect.

Using Data Definition Language (DDL)

DDL statements differ among the IBM database products because storage is handled differently on different systems. On DRDA server systems, there can be several steps between designing a database and issuing a CREATE TABLE statement. For example, a series of statements may translate the design of logical objects into the physical representation of those objects in storage.

The precompiler passes many such DDL statements to the DRDA server when you precompile to a DRDA server database. The same statements would not precompile against a database on the system where the application is running. For example, in an OS/2 application the CREATE STORGROUP statement will precompile successfully to a DB2 for OS/390 database, but not to a DB2 for OS/2 database.

Using Data Manipulation Language (DML)

In general, DML statements are highly portable. SELECT, INSERT, UPDATE, and DELETE statements are similar across the IBM relational database products. Most applications primarily use DML SQL statements, which are supported by the DB2 Connect program and DRDA protocol.

Numeric Data Types

When numeric data is transferred to DB2 Universal Database, the data type may change. Numeric and zoned decimal SQLTYPEs (supported by DB2 for AS/400) are converted to fixed (packed) decimal SQLTYPEs.

Mixed-Byte Data

Mixed-byte data can consist of characters from an extended UNIX code (EUC) character set, a double-byte character set (DBCS) and a single-byte character set (SBCS) in the same column. On systems that store data in EBCDIC (OS/390, OS/400, VSE, and VM), shift-out and shift-in characters mark the start and end of double-byte data. On systems that store data in ASCII (such as OS/2 and UNIX), shift-in and shift-out characters are not required.

If your application transfers mixed-byte data from an ASCII system to an EBCDIC system, be sure to allow enough room for the shift characters. For each switch from SBCS to DBCS data, add 2 bytes to your data length. For better portability, use variable-length strings in applications that use mixed-byte data.

Long Fields

Long fields (strings longer than 254 characters) are handled differently on different systems. A DRDA server may support only a subset of scalar functions for long fields; for example, DB2 for MVS/ESA allows only the **LENGTH** and **SUBSTR** functions for long fields. Also, a DRDA server may require different handling for certain SQL statements; for example, DB2 for VSE & VM requires that with the INSERT statement, only a host variable, the SQLDA, or a NULL value be used.

Large Object (LOB) Data Type

The LOB data type is not supported by the DRDA architecture. DB2 Connect cannot send data with this data type.

Using Data Control Language (DCL)

Each IBM relational database management system provides different levels of granularity for the GRANT and REVOKE SQL statements. Check the product-specific publications to verify the appropriate SQL statements to use for each database management system.

Connecting and Disconnecting

DB2 Connect supports the CONNECT TO and CONNECT RESET versions of the CONNECT statement, as well as CONNECT with no parameters. If an application calls an SQL statement without first performing an explicit CONNECT TO statement, an *implicit* connect is performed to the default application server (if one is defined).

When you connect to a database, information identifying the relational database management system is returned in the SQLERRP field of the SQLCA. If the application server is an IBM relational database, the first three bytes of SQLERRP contain one of the following:

DSN	DB2 for MVS/ESA and DB2 for OS/390
ARI	DB2 for VSE & VM
QSQ	DB2 for AS/400
SQL	DB2 Universal Database.

If you issue a CONNECT TO or null CONNECT statement while using DB2 Connect, the country code or territory token in the SQLERRMC field of the SQLCA is returned as blanks; the CCSID of the application server is returned in the code page or code set token.

You can explicitly disconnect by using the CONNECT RESET statement (for type 1 connect), the RELEASE and COMMIT statements (for type 2 connect), or the DISCONNECT statement (either type of connect, but not in a TP monitor environment). Type 2 connect is used for distributed unit of work, as described in “Distributed Unit of Work” on page 389.

If a connection is not explicitly disconnected and the application ends normally, DB2 Connect commits the resulting data implicitly.

Note: An application can receive SQLCODEs indicating errors and still end normally; DB2 Connect commits the data in this case. If you do not want the data to be committed, you must issue a ROLLBACK command.

The FORCE command lets you disconnect selected users or all users from the database. This is supported for DRDA server databases; the user can be forced off the DB2 Connect workstation.

Precompiling

There are some differences in the precompilers for different IBM relational database systems. The precompiler for DB2 Universal Database differs from the DRDA server precompilers in the following ways:

- It makes only one pass through an application.
- It does not support the DB2 for MVS/ESA DCLGEN command, which is frequently used in application development on MVS.
- When binding against DB2 Universal Database databases, objects must exist for a successful bind. VALIDATE RUN is not supported.

Blocking

The DB2 Connect program supports the DB2 blocking bind options:

UNAMBIG	Only unambiguous cursors are blocked (the default).
ALL	Ambiguous cursors are blocked.
NO	Cursors are not blocked.

The DB2 Connect program uses the block size defined in the DB2 configuration file for the RQRIOBLK field. Current versions of DRDA supports block sizes up to 32 767. If larger values are specified in the DB2 configuration file, DB2 Connect uses a value of 32 767 but does not reset the DB2 configuration file. Blocking is handled the same way using the same block size for dynamic and static SQL.

Note: Most DRDA server systems consider dynamic cursors ambiguous, but DB2 Universal Database systems consider some dynamic cursors unambiguous. To avoid confusion, you can specify BLOCKING ALL with DB2 Connect.

Specify the block size in the DB2 configuration file by using the CLP, the Control Center, or an API, as listed in the *API Reference* and *Command Reference*.

Package Attributes

A package has the following attributes:

Collection ID The ID of the package. It can be specified on the PREP command.

Owner The authorization ID of the package owner. It can be specified on the PREP or BIND command.

Creator The user name that binds the package.

Qualifier The implicit qualifier for objects in the package. It can be specified on the PREP or BIND command.

Each DRDA server system has limitations on the use of these attributes:

DB2 for MVS/ESA and DB2 for OS/390 All four attributes can be different. The use of a different qualifier requires special administrative privileges.

DB2 for VSE & VM All of the attributes must be identical. If USER1 creates a bind file (with PREP), and USER2 performs the actual bind, USER2 needs DBA authority to bind for USER1. Only USER1's user name is used for the attributes.

DB2 for AS/400 The qualifier indicates the collection name. The relationship between qualifiers and ownership affects the granting and revoking of privileges on the object. The user name that is logged on is the creator and owner unless it is qualified by a collection ID, in which case the collection ID is the owner. The collection ID must already exist before it is used as a qualifier.

DB2 Universal Database For DB2 Universal Database databases, the collection ID and the creator can be different. The creator also becomes the owner and the qualifier of the package.

Note: DB2 Connect provides support for the *SET CURRENT PACKAGESET* command for DB2 for MVS/ESA, DB2 for OS/390, and DB2 Universal Database.

C Null-terminated Strings

The CNULREQD bind option overrides the handling of null-terminated strings that are specified using the LANGLEVEL option. See “Null-terminated Strings” on page 435 for a description of how null-terminated strings are handled when prepared with the LANGLEVEL option set to MIA or SAA1. By default, CNULREQD is set to YES. This causes null-terminated strings to be interpreted according to MIA standards. If connecting to a DB2 for OS/390 server it is strongly recommended to set CNULREQD to YES. DB2 Connect will use CNULREQD as a default. You need to bind applications coded to SAA1 standards (with respect to null-terminated strings) with the CNULREQD option set to NO. Otherwise, null-terminated strings will be interpreted according to MIA standards, even if they are prepared using LANGLEVEL set to SAA1.

Standalone SQLCODE and SQLSTATE

Standalone SQLCODE and SQLSTATE variables, as defined in ISO/ANS SQL92, are supported through the LANGLEVEL SQL92E precompile option. An SQL0020W warning will be issued at precompile time, indicating that LANGLEVEL is not supported. This warning applies only to the features listed under LANGLEVEL MIA in the *Command Reference*, which is a subset of LANGLEVEL SQL92E.

Defining a Sort Order

The differences between EBCDIC and ASCII cause differences in sort orders in the various database products, and also affect ORDER BY and GROUP BY clauses. One way to minimize these differences is to create a user-defined collating sequence that mimics the EBCDIC sort order. You can specify a collating sequence only when you create a new database. For more information, see the *API Reference* and the *Command Reference*.

Note: Database tables can now be stored on DB2 for OS/390 in ASCII format. This permits faster exchange of data between DB2 Connect and DB2 for OS/390, and removes the need to provide field procedures which must otherwise be used to convert data and resequence it.

Managing Referential Integrity

Different systems handle referential constraints differently:

DB2 for MVS/ESA or DB2 for OS/390
An index must be created on a primary key before a foreign key can be created using the primary key. Tables can reference themselves.

DB2 for VSE & VM
An index is automatically created for a foreign key. Tables cannot reference themselves.

DB2 for AS/400	An index is automatically created for a foreign key. Tables can reference themselves.
DB2 Universal Database	For DB2 Universal Database databases, an index is automatically created for a foreign key. Tables can reference themselves.

Other rules vary concerning levels of cascade.

Locking

The way in which the database server performs locking can affect some applications. For example, applications designed around row-level locking and the isolation level of cursor stability are not directly portable to systems that perform page-level locking. Because of these underlying differences, applications may need to be adjusted.

The DB2 for OS/390 and DB2 Universal Database products have the ability to time-out a lock and send an error return code to waiting applications.

Differences in SQLCODEs and SQLSTATEs

Different IBM relational database products do not always produce the same SQLCODEs for similar errors. You can handle this problem in either of two ways:

- Use the SQLSTATE instead of the SQLCODE for a particular error.

  SQLSTATEs have approximately the same meaning across the database products, and the products produce SQLSTATEs that correspond to the SQLCODEs.

- Map the SQLCODEs from one system to another system.

  By default, DB2 Connect maps SQLCODEs and tokens from each IBM DRDA server system to your DB2 Universal Database system. You can specify your own SQLCODE mapping file if you want to override the default mapping or you are using a DRDA server that does not have SQLCODE mapping (a non-IBM DRDA server). You can also turn off SQLCODE mapping.

Using System Catalogs

The system catalogs vary across the IBM database products. Many differences can be masked by the use of views. For information, see the documentation for the database server that you are using.

The catalog functions in CLI get around this problem by presenting support of the same API and result sets for catalog queries across the DB2 family (including DRDA).

Numeric Conversion Overflows on Retrieval Assignments

Numeric conversion overflows on retrieval assignments may be handled differently by different IBM relational database products. For example, consider fetching a float column into an integer host variable from DB2 for OS/390 and from DB2 Universal Database. When converting the float value to an integer value, a conversion overflow may occur. By default, DB2 for OS/390 will return a warning sqlcode and a null value to the application. In contrast, DB2 Universal Database will return a conversion overflow error. It is recommended that applications avoid numeric conversion overflows on retrieval assignments by fetching into appropriately sized host variables.

Isolation Levels

DB2 Connect accepts the following isolation levels when you prep or bind an application:

RR	Repeatable Read
RS	Read Stability
CS	Cursor Stability
UR	Uncommitted Read
NC	No Commit

The isolation levels are listed in order from most protection to least protection. If the DRDA server does not support the isolation level that you specify, the next higher supported level is used.

Table 29 on page 549 shows the result of each isolation level on each DRDA application server.

Table 29. Isolation Levels

DB2 Connect	DB2 for MVS/ESA or DB2 for OS/390	DB2 for VSE & VM	DB2 for AS/400	DB2 Universal Database
RR	RR	RR	note 1	RR
RS	note 2	RR	COMMIT(*ALL)	RS
CS	CS	CS	COMMIT(*CS)	CS
UR	note 3	CS	COMMIT(*CHG)	UR
NC	note 4	note 5	COMMIT(*NONE)	UR

Notes:

1. There is no equivalent COMMIT option on DB2 for AS/400 that matches RR. DB2 for AS/400 supports RR by locking the whole table.
2. Results in RR for Version 3.1, and results in RS for Version 4.1 with APAR PN75407 or Version 5.1.
3. Results in CS for Version 3.1, and results in UR for Version 4.1 or Version 5.1.
4. Results in CS for Version 3.1, and results in UR for Version 4.1 with APAR PN60988 or Version 5.1.
5. Isolation level NC is not supported with DB2 for VSE & VM.

With DB2 for AS/400, you can access an unjournalled table if an application is bound with an isolation level of UR and blocking set to ALL, or if the isolation level is set to NC.

Stored Procedures

- Invocation

  A client program can invoke a server program by issuing an SQL CALL statement. Each server works a little differently to the other servers in this case.

  MVS or OS/390 The procedure name must not be more than 8 bytes long and must be defined in the SYSIBM.SYSPROCEDURES catalog on the server.

  VSE or VM Stored procedures are not supported.

  OS/400 The procedure name must be an SQL identifier. You can also use the DECLARE PROCEDURE or CREATE PROCEDURE statements to specify the actual path name (the schema-name or collection-name) to locate the stored procedure.

  All CALL statements to DB2 for AS/400 from REXX/SQL must be dynamically prepared and executed by the application as the CALL statement implemented in REXX/SQL maps to CALL USING DESCRIPTOR.

  For the syntax of the SQL CALL statement, see the *SQL Reference*.

The server program on DB2 Universal Database is invoked with a different parameter convention than a server program on DB2 for MVS/ESA, DB2 for OS/390, or DB2 for AS/400. For more information on the parameter convention for a specific platform, refer to the DB2 product documentation for that platform.

All the SQL statements in a stored procedure are executed as part of the SQL unit of work started by the client SQL program.

- Do not pass indicator values with special meaning to or from stored procedures.

  Between DB2 Universal Database, the systems pass whatever you put into the indicator variables. However, when using the DRDA protocol (such as through DB2 Connect), you can only pass 0, -1, and -128 in the indicator variables.

- You should define a parameter to return any errors or warning encountered by the server application.

  A server program on DB2 Universal Database can update the SQLCA to return any error or warning, but a stored procedure on DB2 for OS/390, DB2 for AS/400, or DB2 for MVS/ESA has no such support. If you want to return an error code from your stored procedure, you must pass it as a parameter. The SQLCODE and SQLCA is only set by the server for system detected errors.

NOT ATOMIC Compound SQL

Compound SQL allows multiple SQL statements to be grouped into a single executable block. This may reduce network overhead and improve response time.

DB2 Connect supports NOT ATOMIC compound SQL. This means that processing of compound SQL continues following an error. (With ATOMIC compound SQL, which is not supported by DB2 Connect, an error would roll back the entire group of compound SQL.)

Statements will continue execution until terminated by the application server. In general, execution of the compound SQL statement will be stopped only in the case of serious errors.

NOT ATOMIC compound SQL can be used with all of the supported DRDA application servers.

If multiple SQL errors occur, the SQLSTATEs of the first seven failing statements are returned in the SQLERRMC field of the SQLCA with a message that multiple errors occurred. For more information, see the *SQL Reference*.

Distributed Unit of Work with DB2 Connect

DB2 Connect allows you to update multiple databases within a single distributed unit of work (DUOW). Whether you can use this capability depends on several factors:

- Your application program must be precompiled with the CONNECT 2 and SYNCPOINT TWOPHASE options.

- Databases which are connected over SNA links (node type APPC) can be updated using two-phase commit (2PC) through DB2 Connect in a single unit of work, *as long as:*
  - The LU 6.2 Syncpoint Manager is installed on AIX or OS/2.
  - The right software prerequisites are installed and configured appropriately: IBM SNA Server for AIX Version 3.1 or higher, or IBM Communications Server for OS/2 Version 4 or higher. (For further information refer to the chapter *Setting Up Two-phase Commit Using SNA* in *DB2 Connect Enterprise Edition Quick Beginnings*).
  - The databases are one of:
    - DB2 for OS/390, Version 5.1 or later
    - DB2 for AS/400, Version 3.1 or later
    - DB2 for MVS/ESA, Version 3.1 or later
    - DB2 for VSE & VM, Version 5.1 or later

    This is regardless of whether CICS or Encina is being used.

    Additionally, the DB2 Connect Enterprise Edition Version 5 Syncpoint Manager continues to allow DB2 MVS Version 3.1 databases to participate in two-phase commit, but *not* to act as the *TM_DATABASE* for such transactions.
- DB2 for OS/390 Version 5 databases connected over TCP/IP links (node type TCPIP) can also participate in units of work using two-phase commit, and they are also able to participate with DB2 Universal Database Version 5 databases in the same transaction. When using TCP/IP, DB2 for OS/390 can be defined as the TM_DATABASE.

Notes:

1. DB2 Common Server Version 2.1 databases can be updated with two-phase commit in a unit of work *only* when DB2 for OS/390 Version 5.1 is *not* the transaction manager database.
2. If the TM_DATABASE for the transaction is DB2 for OS/390 Version 5.1, then DB2 CS V2.1 databases participating in that transaction become *READ-ONLY* for the client application.
3. Mixed connection scenarios for DB2 Universal Database Version 5 client applications are only supported if DB2 for OS/390 Version 5.1 is the *TM_DATABASE* for the transaction. For further information refer to the chapter *Setting Up Two-phase Commit Using TCP/IP* in *DB2 Connect Enterprise Edition Quick Beginnings*.
4. The database levels supported for two-phase commit transactions over TCP/IP depend on the Client Application Enabler level, the TM_DATABASE level, and the participant database levels.

   We strongly recommend that *all* clients accessing DB2 Universal Database Version 5 and DB2 for OS/390 Version 5 databases be at DB2 Universal Database Version 5 level. DB2 Version 2.1 clients *cannot* initiate two-phase commit transactions if DB2 for OS/390 Version 5 databases participate in the transaction.

DRDA Server SQL Statements Supported by DB2 Connect

The following statements compile successfully for DRDA server processing but not for processing with DB2 Universal Database systems:

- ACQUIRE
- DECLARE (modifier.(qualifier.)table_name TABLE ...
- LABEL ON

These statements are also supported by the command line processor.

The following statements are supported for DRDA server processing but are not added to the bind file or the package and are not supported by the command line processor:

- DESCRIBE statement_name INTO descriptor_name USING NAMES
- PREPARE statement_name INTO descriptor_name USING NAMES FROM ...

The precompiler makes the following assumptions:

- Host variables are input variables
- The statement is assigned a unique section number.

DRDA Server SQL Statements Rejected by DB2 Connect

The following SQL statements are not supported by DB2 Connect and not supported by the command line processor:

- COMMIT WORK RELEASE
- DECLARE state_name, statement_name STATEMENT
- DESCRIBE statement_name INTO descriptor_name USING xxxx (where xxxx is ANY, BOTH, or LABELS)
- PREPARE statement_name INTO descriptor_name USING xxxx FROM :host_variable (where xxxx is ANY, BOTH, or LABELS)
- PUT ...
- ROLLBACK WORK RELEASE
- SET :host_variable = CURRENT ...

DB2 for VSE & VM extended dynamic SQL statements are rejected with -104 and syntax error SQLCODEs.

Appendix D. Country Code and Code Page Support

Table 30 shows the languages and code sets supported by the Database Servers and how these values are mapped to country code and code page values that are used by the database manager.

The following is an explanation of each column of the table:

1. **Code Page** shows the IBM-defined code page as mapped from the operating system code set.
2. **Code Set** shows the code set associated with the supported language. The code set is mapped to the DB2 Code Page.
3. **Country Code** shows the country code that is used by the database manager internally for providing country-specific support.
4. **Locale** shows the locale values supported by the database manager.
5. **Operating System** shows the operating system that supports the languages and code sets.

Table 30 (Page 1 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
ARABIC COUNTRIES - AA				
864	IBM-864	785	-	OS/2
1046	IBM-1046	785	Ar_AA	AIX
1089	ISO8859-6	785	ar_AA	AIX
1089	iso88596	785	ar_SA.iso88596	HP
1256	1256	785	-	WIN
420	IBM-420	785	-	HOST
AUSTRALIA - AU				
437	IBM-437	61	-	OS/2
850	IBM-850	61	-	OS/2
819	ISO8859-1	61	en_AU	AIX
850	IBM-850	61	En_AU	AIX
819	iso8859–1	61	-	HP
1051	roman8	61	-	HP
819	ISO8859-1	61	-	Sun
1252	1252	61	-	WIN
1275	1275	61	-	Mac
37	IBM037	61	-	HOST
AUSTRIA - AT				
437	IBM-437	43	-	OS/2

Table 30 (Page 2 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
850	IBM-850	43	-	OS/2
819	ISO8859-1	43	ge_AT	AIX
850	IBM-850	43	Ge_AT	AIX
819	iso88591	43	-	HP
1051	roman8	43	-	HP
819	ISO8859-1	43	-	Sun
1252	1252	43	-	WIN
1275	1275	43	-	Mac
37	IBM-037	43	-	HOST
BELGIUM - BE				
437	IBM-437	32	-	OS/2
850	IBM-850	32	-	OS/2
819	ISO8859-1	32	nl_BE fr_BE	AIX
850	IBM-850	32	Nl_BE Fr_BE	AIX
819	iso88591	32	-	HP
819	ISO8859-1	32	-	Sun
1252	1252	32	-	WIN
1275	1275	32	-	Mac
500	IBM-500	32	-	HOST
BULGARIA - BG				
855	IBM-855	359	-	OS/2
915	ISO8859-5	359	-	OS/2
915	ISO8859-5	359	bg_BG	AIX
915	iso88595	359	-	HP
1251	1251	359	-	WIN
1283	1283	359	-	Mac
1025	IBM-1025	359	-	HOST
BRAZIL - BR				
850	IBM-850	55	-	OS/2
850	IBM-850	55	Pt_BR	AIX
819	ISO8859-1	55	pt_BR	AIX
819	ISO8859-1	55	-	HP
819	ISO8859-1	55	-	Sun
1252	1252	55	-	WIN
37	IBM-037	55	-	HOST

Table 30 (Page 3 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
CANADA - CA				
850	IBM-850	1	-	OS/2
850	IBM-850	1	En_CA	AIX
819	ISO8859-1	1	en_CA	AIX
819	iso88591	1	fr_CA.iso88591	HP
1051	roman8	1	fr_CA.roman8	HP
819	ISO8859-1	1	-	Sun
1252	1252	1	-	WIN
1275	1275	1	-	Mac
37	IBM-037	1	-	HOST
863	IBM-863	2	-	OS/2 (French)
CHINA (PRC) - CN				
1381	IBM-1381	86	-	OS/2
1386	GBK	86	-	OS/2
1383	IBM-eucCN	86	zh_CN	AIX
1386	GBK	86	Zh_CN.GBK	AIX
1383	IBM-eucCN	86	zh_CN.hp15CN	HP
1383	euc-CN	86	zh	Sun
1381	IBM-1381	86	-	WIN
1386	GBK	86	-	WIN
935	IBM-935	86	-	HOST
CROATIA - HR				
852	IBM-852	385	-	OS/2
912	ISO8859-2	385	hr_HR	AIX
912	iso88592	385	hr_HR.iso88592	HP
1250	1250	385	-	WIN
1282	1282	385	-	Mac
870	IBM-870	385	-	HOST
CZECH REPUBLIC - CZ				
852	IBM-852	42	-	OS/2
912	ISO8859-2	42	cs_CZ	AIX
912	iso88592	42	cs_CZ.iso88592	HP
1250	1250	42	-	WIN
1282	1282	42	-	Mac
870	IBM-870	42	-	HOST

Table 30 (Page 4 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
DENMARK - DK				
850	IBM-850	45	-	OS/2
819	ISO8859-1	45	da_DK	AIX
850	IBM-850	45	Da_DK	AIX
819	iso88591	45	da_DK.iso88591	HP
1051	roman8	45	da_DK.roman8	HP
819	ISO8859-1	45	-	Sun
1252	1252	45	-	WIN
1275	1275	45	-	Mac
277	IBM-277	45	-	HOST
ESTONIA - EE				
922	IBM-922	372	-	OS/2
922	IBM-922	372	Et_EE	AIX
922	IBM-922	372	-	WIN
FINLAND - FI				
437	IBM-437	358	-	OS/2
850	IBM-850	358	-	OS/2
819	ISO8859-1	358	fi_FI	AIX
850	IBM-850	358	Fi_FI	AIX
819	iso88591	358	fi_FI.iso88591	HP
819	ISO8859-1	358	-	Sun
1051	roman8	358	-	HP
1252	1252	358	-	WIN
1275	1275	358	-	Mac
278	IBM-278	358	-	HOST
FRANCE - FR				
437	IBM-437	33	-	OS/2
850	IBM-850	33	-	OS/2
819	ISO8859-1	33	fr_FR	AIX
850	IBM-850	33	Fr_FR	AIX
819	iso88591	33	fr_FR.iso88591	HP
1051	roman8	33	fr_FR.roman8	HP
819	ISO8859-1	33	fr	Sun
819	ISO8859-1	33	fr_FR.ISO8859-1	SCO
1252	1252	33	-	WIN

Table 30 (Page 5 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
1275	1275	33	-	Mac
297	IBM-297	33	-	HOST
GERMANY - DE				
437	IBM-437	49	-	OS/2
850	IBM-850	49	-	OS/2
819	ISO8859-1	49	de_DE	AIX
850	IBM-850	49	De_DE	AIX
819	iso88591	49	de_DE.iso88591	HP
1051	roman8	49	de_De.roman8	HP
819	ISO8859-1	49	de	Sun
819	ISO8859-1	49	de_DE.ISO8859-1	SCO
1252	1252	49	-	WIN
1275	1275	49	-	Mac
273	IBM-273	49	-	HOST
819	ISO8859-1	49	De_DE.88591	SINIX
819	ISO8859-1	49	De_DE.6937	SINIX
GREECE - GR				
813	ISO8859-7	30	-	OS/2
869	IBM-869	30	-	OS/2
813	ISO8859-7	30	el_GR	AIX
813	iso8859-7	30	el_GR.iso88597	HP
1253	1253	30	-	WIN
1280	1280	30	-	Mac
423	IBM-423	30	-	HOST
875	IBM-875	30	-	HOST
HUNGARY - HU				
852	IBM-852	36	-	OS/2
912	ISO8859-2	36	hu_HU	AIX
912	iso88592	36	hu_HU.iso88592	HP
1250	1250	36	-	WIN
1282	1282	36	-	Mac
870	IBM-870	36	-	HOST
ICELAND - IS				
850	IBM-850	354	-	OS/2
819	ISO8859-1	354	is_IS	AIX

Table 30 (Page 6 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
850	IBM-850	354	Is_IS	AIX
819	iso88591	354	is_IS.iso88591	HP
1051	roman8	354	is_IS.roman8	HP
819	ISO8859-1	354	-	Sun
1252	1252	354	-	WIN
1275	1275	354	-	Mac
871	IBM-871	354	-	HOST
IRELAND - IE				
437	IBM-437	353	-	OS/2
850	IBM-850	353	-	OS/2
819	ISO8859-1	353	en_IE	AIX
850	IBM-850	353	En_IE	AIX
819	iso88591	353	-	HP
1051	roman8	353	-	HP
819	ISO8859-1	353	-	Sun
819	ISO8859-1	353	en_IE.ISO8859-1	SCO
1252	1252	353	-	WIN
1275	1275	353	-	Mac
285	IBM-285	353	-	HOST
ISRAEL - IL				
862	IBM-862	972	-	OS/2
916	ISO8859-8	972	iw_IL	AIX
1255	1255	972	-	WIN
424	IBM-424	972	-	HOST
ITALY - IT				
437	IBM-437	39	-	OS/2
850	IBM-850	39	-	OS/2
819	ISO8859-1	39	It_IT	AIX
850	IBM-850	39	It_IT	AIX
819	iso88591	39	it_IT.iso88591	HP
1051	roman8	39	it_IT.roman8	HP
819	ISO8859-1	39	it	Sun
1252	1252	39	-	WIN
1275	1275	39	-	Mac
280	IBM-280	39	-	HOST

Table 30 (Page 7 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
JAPAN - JP				
932	IBM-932	81	-	OS/2
942	IBM-942	81	-	OS/2
943	IBM-943	81	-	OS/2
954	IBM-eucJP	81	ja_JP	AIX
932	IBM-932	81	Ja_JP	AIX
954	eucJP	81	ja_JP.eucJP	HP
5039	SJIS	81	ja_JP.SJIS	HP
954	eucJP	81	ja	Sun
943	IBM-943	81	-	WIN
930	IBM-930	81	-	HOST
939	IBM-939	81	-	HOST
5026	IBM-5026	81	-	HOST
5035	IBM-5035	81	-	HOST
LATIN AMERICA - LAT				
437	IBM-437	3	-	OS/2
850	IBM-850	3	-	OS/2
819	ISO8859-1	3	-	AIX
850	IBM-850	3	-	AIX
819	iso88591	3	-	HP
819	ISO8859-1	3	-	Sun
1051	roman8	3	-	HP
1252	1252	3	-	WIN
1275	1275	3	-	Mac
284	IBM-284	3	-	HOST
LATVIA - LV				
921	IBM-921	371	-	OS/2
921	IBM-921	371	Lv_LV	AIX
921	IBM-921	371	-	WIN
LITHUANIA - LT				
921	IBM-921	370	-	OS/2
921	IBM-921	370	Lt_LT	AIX
921	IBM-921	370	-	WIN
MACEDONIA (FYR) - MK				
855	IBM-855	389	-	OS/2

Table 30 (Page 8 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
915	ISO8859-5	389	-	OS/2
915	ISO8859-5	389	mk_MK	AIX
915	iso88595	389	-	HP
1251	1251	389	-	WIN
1283	1283	389	-	Mac
1025	IBM-1025	389	-	HOST
NETHERLANDS - NL				
437	IBM-437	31	-	OS/2
850	IBM-850	31	-	OS/2
819	ISO8859-1	31	nl_NL	AIX
850	IBM-850	31	Nl_NL	AIX
819	iso8859-1	31	nl_NL.iso88591	HP
1051	roman8	31	n1_NL.roman8	HP
819	ISO8859-1	31	nl	Sun
1252	1252	31	-	WIN
1275	1275	31	-	Mac
37	IBM-037	31	-	HOST
NEW ZEALAND - NZ				
850	IBM-850	64	-	OS/2
850	IBM-850	64	En_NZ	AIX
819	ISO8859-1	64	en_NZ	AIX
819	ISO8859-1	64	-	HP
819	ISO8859-1	64	-	Sun
1252	1252	64	-	WIN
37	IBM-037	64	-	HOST
NORWAY - NO				
850	IBM-850	47	-	OS/2
819	ISO8859-1	47	no_NO	AIX
850	IBM-850	47	No_NO	AIX
819	iso88591	47	no_NO.iso88591	HP
1051	roman8	47	no_NO.roman8	HP
819	ISO8859-1	47	-	Sun
1252	1252	47	-	WIN
1275	1275	47	-	Mac
277	IBM-277	47	-	HOST

Table 30 (Page 9 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
POLAND - PL				
852	IBM-852	48	-	OS/2
912	ISO8859-2	48	pl_PL	AIX
912	iso8859-2	48	pl_PL.iso88592	HP
1250	1250	48	-	WIN
1282	1282	48	-	Mac
870	IBM-870	48	-	HOST
PORTUGAL - PT				
860	IBM-860	351	-	OS/2
850	IBM-850	351	-	OS/2
819	ISO8859-1	351	pt_PT	AIX
850	IBM-850	351	Pt_PT	AIX
819	iso8859-1	351	pt_PT.iso88591	HP
1051	roman8	351	pt_PT.roman8	HP
819	ISO8859-1	351	-	Sun
1252	1252	351	-	WIN
1275	1275	351	-	Mac
37	IBM-037	351	-	HOST
ROMANIA - RO				
852	IBM-852	40	-	OS/2
912	ISO8859-2	40	ro_RO	AIX
912	iso8859-2	40	ro_RO.iso88592	HP
1250	1250	40	-	WIN
1282	1282	40	-	Mac
870	IBM-870	40	-	HOST
RUSSIA - RU				
866	IBM-866	7	-	OS/2
915	ISO8859-5	7	-	OS/2
915	ISO8859-5	7	ru_RU	AIX
915	iso8859-5	7	ru_RU.iso88585	HP
1251	1252	7	-	WIN
1283	1283	7	-	Mac
1025	IBM-1025	7	-	HOST
SERBIA/MONTENEGRO - SP				
855	IBM-855	381	-	OS/2

Table 30 (Page 10 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
915	ISO8859-5	381	-	OS/2
915	ISO8859-5	381	sr_SP	AIX
915	iso8859-5	381	-	HP
1251	1251	381	-	WIN
1283	1283	381	-	Mac
1025	IBM-1025	381	-	HOST
SLOVAKIA - SK				
852	IBM-852	938	-	OS/2
912	ISO8859-2	938	sk_SK	AIX
912	iso8859-2	938	sk_SK.iso88592	HP
1250	1250	938	-	WIN
1282	1282	938	-	Mac
870	IBM-870	938	-	HOST
SLOVENIA - SI				
852	IBM-852	386	-	OS/2
912	ISO8859-2	386	sl_SL	AIX
912	iso8859-2	386	sl_SL.iso88592	HP
1250	1250	386	-	WIN
1282	1282	386	-	Mac
870	IBM-870	386	-	HOST
SOUTH AFRICA - ZA				
437	IBM-437	227	-	OS/2
850	IBM-850	27	-	OS/2
819	ISO8859-1	27	en_ZA	AIX
850	IBM-850	27	En_ZA	AIX
819	iso8859-1	27	-	HP
1051	roman8	27	-	HP
819	ISO8859-1	27	-	Sun
819	ISO8859-1	27	en_ZA.ISO8859-1	SCO
1252	1252	27	-	WIN
1275	1275	27	-	Mac
285	IBM-285	27	-	HOST
SOUTH KOREA - KR				
949	IBM-949	82	-	OS/2
970	IBM-eucKR	82	ko_KR	AIX

Table 30 (Page 11 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
970	eucKR	82	ko_KR.eucKR	HP
970	eucKR	82	ko	Sun
1363	1363	82	-	WIN
933	IBM-933	82	-	HOST
SPAIN - ES				
437	IBM-437	34	-	OS/2
850	IBM-850	34	-	OS/2
819	ISO8859-1	34	es_ES	AIX
850	IBM-850	34	Es_ES	AIX
819	iso8859-1	34	es_ES.iso88591	HP
1051	roman8	34	es_ES.roman8	HP
819	ISO8859-1	34	es	Sun
819	ISO8859-1	34	es_ES.ISO8859-1	SCO
1252	1252	34	-	WIN
1275	1275	34	-	Mac
284	IBM-284	34	-	HOST
SWEDEN - SE				
437	IBM-437	46	-	OS/2
850	IBM-850	46	-	OS/2
819	ISO8859-1	46	sv_SE	AIX
850	IBM-850	46	Sv_SE	AIX
819	iso88591	46	sv_SE.iso88591	HP
1051	roman8	46	sv_SE.roman8	HP
819	ISO8859-1	46	sv	Sun
1252	1252	46	-	WIN
1275	1275	46	-	Mac
278	IBM-278	46	-	HOST
SWITZERLAND - CH				
437	IBM-437	41	-	OS/2
850	IBM-850	41	-	OS/2
819	ISO8859-1	41	de_CH	AIX
850	IBM-850	41	De_CH	AIX
819	iso88591	41	-	HP
1051	roman8	41	-	HP
819	ISO8859-1	41	-	Sun

Table 30 (Page 12 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
1252	1252	41	-	WIN
1275	1275	41	-	Mac
500	IBM-500	41	-	HOST
TAIWAN - TW				
938	IBM-938	886	-	OS/2
948	IBM-948	886	-	OS/2
950	big5	886	-	OS/2
950	big5	886	Zh_TW	AIX
964	IBM-eucTW	886	Zh_TW	AIX
950	big5	886	zh_TW.big5	HP
964	eucTW	886	zh_TW.eucTW	HP
950	big5	886	big5	Sun
964	eucTW	886	zh_TW	Sun
950	big5	886	-	Win
937	IBM-937	886	-	HOST
THAILAND - TH				
874	TIS620-1	66	-	OS/2
874	TIS620-1	66	Th_TH	AIX
874	tis620	66	th_TH.tis620	HP
874	TIS620-1	66	-	WIN
838	IBM-838	66	-	HOST
TURKEY - TR				
857	IBM-857	90	-	OS/2
920	ISO8859-9	90	tr_TR	AIX
920	ISO8859-9	90	tr_TR.iso88599	HP
1254	1254	90	-	WIN
1281	1281	90	-	Mac
1026	IBM-1026	90	-	HOST
UNITED KINGDOM - GB				
437	IBM-437	44	-	OS/2
850	IBM-850	44	-	OS/2
819	ISO8859-1	44	en_GB	AIX
850	IBM-850	44	En_GB	AIX
819	iso88591	44	en_GB.iso88591	HP
1051	roman8	44	en_GB.roman8	HP

Table 30 (Page 13 of 13). Supported Languages and Code Sets

Code Page	Code Set	Ctry. Code	Locale	Op. System
819	ISO8859-1	44	-	Sun
819	ISO8859-1	44	en_GB.ISO8859-1	SCO
819	88591	44	En_GB.88591	SINIX
819	ISO8859-1	44	En_GB.6937	SINIX
1252	1252	44	-	WIN
1275	1275	44	-	Mac
285	IBM-285	44	-	HOST
United States of America - US				
437	IBM-437	1	-	OS/2
850	IBM-850	1	-	OS/2
819	ISO8859-1	1	en_US	AIX
850	IBM-850	1	En_US	AIX
819	iso8859-1	1	en_US.iso88591	HP
1051	roman8	1	en_US.roman8	HP
819	ISO8859-1	1	en_US	Sun
819	ISO8859-1	1	en_US	SGI
819	ISO8859-1	1	en_US.ISO8859-1	SCO
1252	1252	1	-	WIN
1275	1275	1	-	Mac
37	IBM-037	1	-	HOST

Note: The Solaris code page 950 does not support the following characters in IBM 950:

Code Range	Description	Sun Big-5	IBM Big-5
C6A1-C8FE	Symbols	Reserved area	Symbols
F9D6-F9FE	ETen extension	Reserved area	ETen extension
F286-F9A0	IBM selected chars	Reserved area	IBM selected

Appendix E. Simulating EBCDIC Binary Collation

With DB2, you can collate character strings according to a user-defined collating sequence. You can use this feature to simulate EBCDIC binary collation.

As an example of how to simulate EBCDIC collation, suppose you want to create an ASCII database with code page 850, but you also want the character strings to be collated as if the data actually resides in an EBCDIC database with code page 500. See Figure 54 on page 570 for the definition of code page 500, and Figure 55 on page 571 for the definition of code page 850.

Consider the relative collation of four characters in a EBCDIC code page 500 database, when they are collated in binary:

Character	**Code Page 500 Code Point**
'a'	X'81'
'b'	X'82'
'A'	X'C1'
'B'	X'C2'

The code page 500 binary collation sequence (the desired sequence) is:

```
'a' < 'b' < 'A' < 'B'
```

If you create the database with ASCII code page 850, binary collation would yield:

Character	**Code Page 850 Code Point**
'a'	X'61'
'b'	X'62'
'A'	X'41'
'B'	X'42'

The code page 850 binary collation (which is not the desired sequence) is:

```
'A' < 'B' < 'a' < 'b'
```

To achieve the desired collation, you need to create your database with a user-defined collating sequence. A sample collating sequence for just this purpose is supplied with DB2 in the `sqle850a.h` include file. The content of `sqle850a.h` is shown in Figure 53 on page 568.

```
#ifndef SQL_H_SQLE850A
#define SQL_H_SQLE850A

#ifdef __cplusplus
extern "C" {
#endif

unsigned char sqle_850_500[256] = {
0x00,0x01,0x02,0x03,0x37,0x2d,0x2e,0x2f,0x16,0x05,0x25,0x0b,0x0c,0x0d,0x0e,0x0f,
0x10,0x11,0x12,0x13,0x3c,0x3d,0x32,0x26,0x18,0x19,0x3f,0x27,0x1c,0x1d,0x1e,0x1f,
0x40,0x4f,0x7f,0x7b,0x5b,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
0x7c, 0xc1 , 0xc2 ,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,
0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0xe0,0x5a,0x5f,0x6d,
0x79, 0x81 , 0x82 ,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,0x95,0x96,
0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0xbb,0xd0,0xa1,0x07,
0x68,0xdc,0x51,0x42,0x43,0x44,0x47,0x48,0x52,0x53,0x54,0x57,0x56,0x58,0x63,0x67,
0x71,0x9c,0x9e,0xcb,0xcc,0xcd,0xdb,0xdd,0xdf,0xec,0xfc,0x70,0xb1,0x80,0xbf,0xff,
0x45,0x55,0xce,0xde,0x49,0x69,0x9a,0x9b,0xab,0xaf,0xba,0xb8,0xb7,0xaa,0x8a,0x8b,
0x2b,0x2c,0x09,0x21,0x28,0x65,0x62,0x64,0xb4,0x38,0x31,0x34,0x33,0xb0,0xb2,0x24,
0x22,0x17,0x29,0x06,0x20,0x2a,0x46,0x66,0x1a,0x35,0x08,0x39,0x36,0x30,0x3a,0x9f,
0x8c,0xac,0x72,0x73,0x74,0x0a,0x75,0x76,0x77,0x23,0x15,0x14,0x04,0x6a,0x78,0x3b,
0xee,0x59,0xeb,0xed,0xcf,0xef,0xa0,0x8e,0xae,0xfe,0xfb,0xfd,0x8d,0xad,0xbc,0xbe,
0xca,0x8f,0x1b,0xb9,0xb6,0xb5,0xe1,0x9d,0x90,0xbd,0xb3,0xda,0xfa,0xea,0x3e,0x41
};
#ifdef __cplusplus
}
#endif

#endif /* SQL_H_SQLE850A */
```

Figure 53. User-Defined Collating Sequence - sqle_850_500

To see how to achieve code page 500 binary collation on code page 850 characters, examine the sample collating sequence in `sqle_850_500`. For each code page 850 character, its weight in the collating sequence is simply its corresponding code point in code page 500.

For example, consider the letter ‘a’. This letter is code point X'61' for code page 850 as shown in Figure 55 on page 571. In the array `sqle_850_500`, letter ‘a’ is assigned a weight of X'81' (that is, the 98th element in the array `sqle_850_500`).

Consider how the four characters collate when the database is created with the above sample user-defined collating sequence:

Character	Code Page 850 Code Point / Weight (from sqle_850_500)
'a'	X'61' / X'81'
'b'	X'62' / X'82'
'A'	X'41' / X'C1'
'B'	X'42' / X'C2'

The code page 850 user-defined collation by weight (the desired collation) is:

```
'a' < 'b' < 'A' < 'B'
```

In this example, you achieve the desired collation by specifying the correct weights to simulate the desired behavior.

Closely observing the actual collating sequence, notice that the sequence itself is merely a conversion table, where the source code page is the code page of the data base (850) and the target code page is the desired binary collating code page (500). Other sample collating sequences supplied by DB2 enable different conversions. If a conversion table that you require is not supplied with DB2, additional conversion tables can be obtained from the IBM publication, *Character Data Representation Architecture, Reference and Registry*, SC09-2190. You will find the additional conversion tables in a CD-ROM enclosed with this publication.

For more details on collating sequences, see “Collating Sequences” on page 365. Also see the CREATE DATABASE API described in the *API Reference* for a description of the collating sequences supplied with DB2, and for the listing of a sample program (`db_udcs.c`) that demonstrates how to create a database with a user-defined collating sequence.

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¢ SC040000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	[SM060000	] SM080000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	¹ ND011000	² ND021000	³ ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	| SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 00500

Figure 54. Code Page 500

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	Ç LC420000	É LE120000	á LA110000	░ SF140000	└ SF020000	ð LD630000	Ó LO120000	(SHY) SP320000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	▒ SF150000	┴ SF070000	Ð LD620000	ß LS610000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	▓ SF160000	┬ SF060000	Ê LE160000	Ô LO160000	‗ SM100000
-3	♥ SS020000	‼ SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	│ SF110000	├ SF080000	Ë LE180000	Ò LO140000	¾ NF050000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	ñ LN190000	┤ SF090000	─ SF100000	È LE140000	õ LO190000	¶ SM250000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	ò LO130000	Ñ LN200000	Á LA120000	┼ SF050000	ı LI610000	Õ LO200000	§ SM240000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	ª SM210000	Â LA160000	ã LA190000	Í LI120000	µ SM170000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ù LU130000	º SM200000	À LA140000	Ã LA200000	Î LI160000	þ LT630000	¸ SD410000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY170000	¿ SP160000	© SM520000	╚ SF380000	Ï LI180000	Þ LT640000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	® SM530000	╣ SF230000	╔ SF390000	┘ SF040000	Ú LU120000	¨ SD170000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	║ SF240000	╩ SF400000	┌ SF010000	Û LU160000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	ï LI170000	ø LO610000	½ NF010000	╗ SF250000	╦ SF410000	█ SF610000	Ù LU140000	¹ ND011000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	î LI150000	£ SC020000	¼ NF040000	╝ SF260000	╠ SF420000	▄ SF570000	ý LY110000	³ ND031000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000	ì LI130000	Ø LO620000	¡ SP030000	¢ SC040000	═ SF430000	¦ SM650000	Ý LY120000	² ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	× SA070000	« SP170000	¥ SC050000	╬ SF440000	Ì LI140000	¯ SM150000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Å LA280000	ƒ SC070000	» SP180000	┐ SF030000	¤ SC010000	▀ SF600000	´ SD110000	(RSP) SP300000

Code Page 00850

Figure 55. Code Page 850

Appendix F. How the DB2 Library Is Structured

The DB2 Universal Database library consists of SmartGuides, online help, and books. This section describes the information that is provided, and how to access it.

To help you access product information online, DB2 provides the Information Center on OS/2, Windows 95, and the Windows NT operating systems. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web. "About the Information Center" on page 580 has more details.

SmartGuides

SmartGuides help you complete some administration tasks by taking you through each task one step at a time. SmartGuides are available on OS/2, Windows 95, and the Windows NT operating systems. The following table lists the SmartGuides.

SmartGuide	Helps you to...	How to Access...
Add Database	Catalog a database on a client workstation.	From the Client Configuration Assistant, click on **Add**.
Create Database	Create a database, and to perform some basic configuration tasks.	From the Control Center, click with the right mouse button on the **Databases** icon and select **Create**->**New**.
Performance Configuration	Tune the performance of a database by updating configuration parameters to match your business requirements.	From the Control Center, click with the right mouse button on the database you want to tune and select **Configure performance**.
Backup Database	Determine, create, and schedule a backup plan.	From the Control Center, click with the right mouse button on the database you want to backup and select **Backup**->**Database using SmartGuide**.
Restore Database	Recover a database after a failure. It helps you understand which backup to use, and which logs to replay.	From the Control Center, click with the right mouse button on the database you want to restore and select **Restore**->**Database using SmartGuide**.
Create Table	Select basic data types, and create a primary key for the table.	From the Control Center, click with the right mouse button on the **Tables** icon and select **Create**->**Table using SmartGuide**.
Create Table Space	Create a new table space.	From the Control Center, click with the right mouse button on the **Table spaces** icon and select **Create**->**Table space using SmartGuide**.

Online Help

Online help is available with all DB2 components. The following table describes the various types of help.

Type of Help	Contents	How to Access...
Command Help	Explains the syntax of commands in the command line processor.	From the command line processor in interactive mode, enter: **?** *command* where *command* is a keyword or the entire command. For example, **?** *catalog* displays help for all the CATALOG commands, whereas **?** *catalog database* displays help for the CATALOG DATABASE command.
Control Center Help	Explains the tasks you can perform in a window or notebook. The help includes prerequisite information you need to know, and describes how to use the window or notebook controls.	From a window or notebook, click on the **Help** push button or press the F1 key.
Message Help	Describes the cause of a message number, and any action you should take.	From the command line processor in interactive mode, enter: **?** *message number* where *message number* is a valid message number. For example, **?** *SQL30081* displays help about the SQL30081 message. To view message help one screen at a time, enter: **?** *XXXnnnnn* **| more** where *XXX* is the message prefix, such as SQL, and *nnnnn* is the message number, such as 30081. To save message help in a file, enter: **?** *XXXnnnnn* > *filename.ext* where *filename.ext* is the file where you want to save the message help. **Note:** On UNIX-based systems, enter: **\?** *XXXnnnnn* **| more** or **\?** *XXXnnnnn* > *filename.ext*

Type of Help	Contents	How to Access...
SQL Help	Explains the syntax of SQL statements.	From the command line processor in interactive mode, enter: **help** *statement* where *statement* is an SQL statement. For example, **help** *SELECT* displays help about the SELECT statement.
SQLSTATE Help	Explains SQL states and class codes.	From the command line processor in interactive mode, enter: **?** *sqlstate* or **?** *class-code* where *sqlstate* is a valid five digit SQL state and *class-code* is a valid two digit class code. For example, **?** *08003* displays help for the 08003 SQL state, whereas **?** *08* displays help for the 08 class code.

DB2 Books

The table in this section lists the DB2 books. They are divided into two groups:

- Cross-platform books: These books are for DB2 on any of the supported platforms.
- Platform-specific books: These books are for DB2 on a specific platform. For example, there is a separate *Quick Beginnings* book for DB2 on OS/2, Windows NT, and UNIX-based operating systems.

Most books are available in HTML and PostScript format, and in hardcopy that you can order from IBM. The exceptions are noted in the table.

You can obtain DB2 books and access information in a variety of different ways:

View To view an HTML book, you can do the following:

- If you are running DB2 administration tools on OS/2, Windows 95, or the Windows NT operating systems, you can use the Information Center. “About the Information Center” on page 580 has more details.
- Use the open file function of the Web browser supplied by DB2 (or one of your own) to open the following page:

  sqllib/doc/html/index.htm

  The page contains descriptions of and links to the DB2 books. The path is located on the drive where DB2 is installed.

  You can also open the page by double-clicking on the **DB2 Online Books** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

Search To search for information in the HTML books, you can do the following:

- Click on **Search the DB2 Books** at the bottom of any page in the HTML books. Use the search form to find a specific topic.
- Click on **Index** at the bottom of any page in an HTML book. Use the Index to find a specific topic in the book.
- Display the Table of Contents or Index of the HTML book, and then use the find function of the Web browser to find a specific topic in the book.
- Use the bookmark function of the Web browser to quickly return to a specific topic.
- Use the search function of the Information Center to find specific topics. “About the Information Center” on page 580 has more details.

Print To print a book on a PostScript printer, look for the file name shown in the table.

Order To order a hardcopy book from IBM, use the form number.

Book Name	Book Description	Form Number File Name
	Cross-Platform Books	
Administration Getting Started	Introduces basic DB2 database administration concepts and tasks, and walks you through the primary administrative tasks.	S10J-8154 db2k0x50
Administration Guide	Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.	S10J-8157 db2d0x50
API Reference	Describes the DB2 application programming interfaces (APIs) and data structures you can use to manage your databases. Explains how to call APIs from your applications.	S10J-8167 db2b0x50
CLI Guide and Reference	Explains how to develop applications that access DB2 databases using the DB2 Call Level Interface, a callable SQL interface that is compatible with the Microsoft ODBC specification.	S10J-8159 db2l0x50
Command Reference	Explains how to use the command line processor, and describes the DB2 commands you can use to manage your database.	S10J-8166 db2n0x50
DB2 Connect Enterprise Edition Quick Beginnings	Provides planning, installing, configuring, and using information for DB2 Connect Enterprise Edition. Also contains installation and setup information for all supported clients.	S10J-7888 db2cyx50
DB2 Connect Personal Edition Quick Beginnings	Provides planning, installing, configuring, and using information for DB2 Connect Personal Edition.	S10J-8162 db2c1x50
DB2 Connect User's Guide	Provides concepts, programming and general using information about the DB2 Connect products.	S10J-8163 db2c0x50
DB2 Connectivity Supplement	Provides setup and reference information for customers who want to use DB2 for AS/400, DB2 for OS/390, DB2 for MVS, or DB2 for VM as DRDA Application Requesters with DB2 Universal Database servers, and customers who want to use DRDA Application Servers with DB2 Connect (formerly DDCS) application requesters. **Note:** Available in HTML and PostScript formats only.	No form number db2h1x50
Embedded SQL Programming Guide	Explains how to develop applications that access DB2 databases using embedded SQL, and includes discussions about programming techniques and performance considerations.	S10J-8158 db2a0x50
Glossary	Provides a comprehensive list of all DB2 terms and definitions. **Note:** Available in HTML format only.	No form number db2t0x50

Book Name	Book Description	Form Number File Name
Installing and Configuring DB2 Clients	Provides installation and setup information for all DB2 Client Application Enablers and DB2 Software Developer's Kits. **Note:** Available in HTML and PostScript formats only.	No form number db2iyx50
Master Index	Contains a cross reference to the major topics covered in the DB2 library. **Note:** Available in PostScript format and hardcopy only.	S10J-8170 db2w0x50
Message Reference	Lists messages and codes issued by DB2, and describes the actions you should take.	S10J-8168 db2m0x50
Replication Guide and Reference	Provides planning, configuring, administering, and using information for the IBM Replication tools supplied with DB2.	S95H-0999 db2e0x50
Road Map to DB2 Programming	Introduces the different ways your applications can access DB2, describes key DB2 features you can use in your applications, and points to detailed sources of information for DB2 programming.	S10J-8155 db2u0x50
SQL Getting Started	Introduces SQL concepts, and provides examples for many constructs and tasks.	S10J-8156 db2y0x50
SQL Reference	Describes SQL syntax, semantics, and the rules of the language. Also includes information about release-to-release incompatibilities, product limits, and catalog views.	S10J-8165 db2s0x50
System Monitor Guide and Reference	Describes how to collect different kinds of information about your database and the database manager. Explains how you can use the information to understand database activity, improve performance, and determine the cause of problems.	S10J-8164 db2f0x50
Troubleshooting Guide	Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service.	S10J-8169 db2p0x50
What's New	Describes the new features, functions, and enhancements in DB2 Universal Database. **Note:** Available in HTML and PostScript formats only.	No form number db2q0x50
Platform-Specific Books		
Building Applications for UNIX Environments	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a UNIX system.	S10J-8161 db2axx50
Building Applications for Windows and OS/2 Environments	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a Windows or OS/2 system.	S10J-8160 db2a1x50

Book Name	Book Description	Form Number File Name
DB2 Extended Enterprise Edition Quick Beginnings	Provides planning, installing, configuring, and using information for DB2 Universal Database Extended Enterprise Edition for AIX.	S72H-9620 db2v3x50
DB2 Personal Edition Quick Beginnings	Provides planning, installing, configuring, and using information for DB2 Universal Database Personal Edition on OS/2, Windows 95, and the Windows NT operating systems.	S10J-8150 db2i1x50
DB2 SDK for Macintosh Building Your Applications	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a Macintosh system. **Note:** Available in PostScript format and hardcopy for DB2 Version 2.1.2 only.	S50H-0528 sqla7x02
DB2 SDK for SCO OpenServer Building Your Applications	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a SCO OpenServer system. **Note:** Available for DB2 Version 2.1.2 only.	S89H-3242 sqla9x02
DB2 SDK for Silicon Graphics IRIX Building Your Applications	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a Silicon Graphics system. **Note:** Available in PostScript format and hardcopy for DB2 Version 2.1.2 only.	S89H-4032 sqlaax02
DB2 SDK for SINIX Building Your Applications	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a SINIX system. **Note:** Available in PostScript format and hardcopy for DB2 Version 2.1.2 only.	S50H-0530 sqla8x00
Quick Beginnings for OS/2	Provides planning, installing, configuring, and using information for DB2 Universal Database on OS/2. Also contains installing and setup information for all supported clients.	S10J-8147 db2i2x50
Quick Beginnings for UNIX	Provides planning, installing, configuring, and using information for DB2 Universal Database on UNIX-based platforms. Also contains installing and setup information for all supported clients.	S10J-8148 db2ixx50
Quick Beginnings for Windows NT	Provides planning, installing, configuring, and using information for DB2 Universal Database on the Windows NT operating system. Also contains installing and setup information for all supported clients.	S10J-8149 db2i6x50

Notes:

1. The character in the sixth position of the file name indicates the language of a book. For example, the file name `db2d0e50` indicates that the *Administration Guide* is in English. The following letters are used in the file names to indicate the language of a book:

Language	Identifier	Language	Identifier
Brazilian Portuguese	B	Hungarian	H
Bulgarian	U	Italian	I
Czech	X	Norwegian	N
Danish	D	Polish	P
English	E	Russian	R
Finnish	Y	Slovenian	L
French	F	Spanish	Z
German	G	Swedish	S

2. For late breaking information that could not be included in the DB2 books, see the README file. Each DB2 product includes a README file which you can find in the directory where the product is installed.

About the Information Center

The Information Center provides quick access to DB2 product information. The Information Center is available on OS/2, Windows 95, and the Windows NT operating systems. You must install the DB2 administration tools to see the Information Center.

Depending on your system, you can access the Information Center from the:

- Main product folder
- Toolbar in the Control Center
- Windows Start menu.

The Information Center provides the following kinds of information. Click on the appropriate tab to look at the information:

Tasks Lists tasks you can perform using DB2.

Reference Lists DB2 reference information, such as keywords, commands, and APIs.

Books Lists DB2 books.

Troubleshooting Lists categories of error messages and their recovery actions.

Sample Programs Lists sample programs that come with the DB2 Software Developer's Kit. If the Software Developer's Kit is not installed, this tab is not displayed.

Web Lists DB2 information on the World Wide Web. To access this information, you must have a connection to the Web from your system.

When you select an item in one of the lists, the Information Center launches a viewer to display the information. The viewer might be the system help viewer, an editor, or a Web browser, depending on the kind of information you select.

The Information Center provides search capabilities so you can look for specific topics, and filter capabilities to limit the scope of your searches.

Appendix G. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the

IBM Director of Licensing,
IBM Corporation,
500 Columbus Avenue,
Thornwood, NY, 10594
USA.

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 and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Department 071
1150 Eglinton Ave. East
North York, Ontario
M3C 1H7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

This publication may contain 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.

Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

ACF/VTAM
ADSTAR
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
CICS
C Set++
C/370
DATABASE 2
DatagLANce
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
Distributed Relational Database Architecture
DRDA
Extended Services
FFST
First Failure Support Technology
IBM
IMS
Lan Distance
MVS/ESA
MVS/XA
NetView
OS/400
OS/390
OS/2
PowerPC
QMF
RACF
RISC System/6000
SAA
SP
SQL/DS
SQL/400
S/370
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WIN-OS/2

Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc.

HP-UX is a trademark of Hewlett-Packard.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Solaris is a trademark of Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, or service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Index

Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

Telephone

If you live in the U.S.A., call one of the following numbers:

- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by selecting the "Roadmap to IBM Support" item at: http://www.ibm.com/support/.

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

World Wide Web
http://www.software.ibm.com/data/
http://www.software.ibm.com/data/db2/library/

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

Anonymous FTP Sites
ftp.software.ibm.com

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools concerning DB2 and many related products.

Internet Newsgroups
comp.databases.ibm-db2, bit.listserv.db2-l

These newsgroups are available for users to discuss their experiences with DB2 products.

CompuServe
GO IBMDB2 to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to http://www.software.ibm.com/data/db2/db2tech/db2cert.html

Part Number: 10J8158

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

10J8158

S10J-8158-00