IBM® DB2® Universal Database

Application Development Guide

Version 7

SC09-2949-01

IBM® DB2® Universal Database

Application Development Guide

Version 7

SC09-2949-01

Before using this information and the product it supports, be sure to read the general information under "Appendix G. Notices" on page 827.

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

Contents

Part 1. DB2 Application Development Concepts

Chapter 1. Getting Started with DB2 Application Development

About This Book

This book discusses how to design and code application programs that access DB2 databases. It presents detailed information on the use of Structured Query Language (SQL) in supported host language programs. For information on language support for your specific operating system, see the *Application Building Guide*. This book also provides an overview of some of the DB2 utilities that you can use to help create DB2 applications. These utilities include "The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 30 and "Chapter 9. IBM DB2 Stored Procedure Builder" on page 269.

You can access data with:

- SQL statements embedded in a host language, including embedded SQL for Java (SQLJ)
- dynamic APIs including Java Database Connectivity (JDBC), Perl DBI, and DB2 Call Level Interface (DB2 CLI)

This book discusses all these ways to access data except DB2 CLI, which is discussed in the *CLI Guide and Reference*. JDBC, SQLJ, and DB2 CLI provide some data access capabilities that are not available through embedded SQL. These capabilities include scrollable cursors and stored procedures that return multiple result sets. See the discussion in "Access to Data" on page 23 to help you decide which data access method to use.

To effectively use the information in this book to design, write, and test your DB2 application programs, you need to refer to the *SQL Reference* along with this book. If you are using the DB2 Call Level Interface (CLI) or Open Database Connectivity (ODBC) interface in your applications to access DB2 databases, refer to the *CLI Guide and Reference*. To perform database manager administration functions using the DB2 administration APIs in your application programs, refer to the *Administrative API Reference*.

You can also develop applications where one part of the application runs on the client and another part runs on the server. Version 7 of DB2 introduces

support for stored procedures with enhanced portability and scalability across platforms. Stored procedures are discussed in "Chapter 7. Stored Procedures" on page 193.

You can use object-based extensions to DB2 to make your DB2 application programs more powerful, flexible, and active than traditional DB2 applications. The extensions include large objects (*LOBs*), distinct types, structured types, user-defined functions (*UDFs*), and triggers. These features of DB2 are described in:

- "Chapter 10. Using the Object-Relational Capabilities" on page 275
- "Chapter 11. User-defined Distinct Types" on page 281
- "Chapter 12. Working with Complex Objects: User-Defined Structured Types" on page 291
- "Chapter 13. Using Large Objects (LOBs)" on page 349
- "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373
- "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393
- "Chapter 16. Using Triggers in an Active DBMS" on page 483

References to DB2 in this book should be understood to mean the DB2 Universal Database product on UNIX, Linux, OS/2, and Windows 32-bit operating systems. References to DB2 on other platforms use a specific product name and platform, such as DB2 Universal Database for AS/400.

Who Should Use This Book

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

How to Use This Book

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

- Part 1. DB2 Application Development Concepts contains information you need to use this book and an overview of the methods you can use to develop applications for DB2 Universal Database.
  - Chapter 1. Getting Started with DB2 Application Development describes the structure of this book and the conventions used in it.
  - Chapter 2. Coding a DB2 Application introduces the overall application development process using DB2. It discusses and compares the important application design issues you need to consider prior to coding

your applications. This chapter concludes with information to help you set up a test environment where you can begin to develop your applications.

- Part 2. Embedding SQL in Applications describes how to embed static and dynamic SQL in your applications. This information includes a description of the utilities that you can use to help create your embedded SQL applications.
  - Embedding SQL Statements in a Host Language discusses the process of creating a DB2 application by embedding SQL in host languages such as C/C++, Java, and COBOL. It contains an overview of the DB2 precompiler, compiling and linking the application, and binding the embedded SQL statements to the database.
  - Chapter 4. Writing Static SQL Programs discusses the details of coding your DB2 embedded SQL application using static SQL statements. It contains detailed guidelines and considerations for using static SQL.
  - Chapter 5. Writing Dynamic SQL Programs discusses the details of coding your DB2 embedded SQL application using dynamic SQL statements. It contains detailed guidelines and considerations for using dynamic SQL.
  - Chapter 6. Common DB2 Application Techniques discusses DB2 features that help you with common application development problems. These features include the ability to automatically create unique row identifiers, to create columns that are dynamically derived from an expression, and to create and use declared temporary tables.
- Part 3. Stored Procedures discusses how to use stored procedures to improve the performance of database applications that run in client/server environments.
  - Chapter 7. Stored Procedures describes how to write stored procedures and the client applications that call stored procedures using host languages.
  - Chapter 8. Writing SQL Procedures describes how to write stored procedures in SQL by issuing a CREATE PROCEDURE statement. SQL procedures encode their procedural logic using SQL in the body of the CREATE PROCEDURE statement.
  - Chapter 9. IBM DB2 Stored Procedure Builder describes the IBM DB2 Stored Procedure Builder, a graphical application that supports the rapid development of stored procedures for DB2. Stored Procedure Builder helps you create both SQL and Java stored procedures.
- Part 4. Object-Relational Programming describes how to use the object-relational support provided by DB2. This information includes an introduction to and detailed instructions on how to use large objects, user-defined functions, user-defined distinct types, and triggers.

- Chapter 10. Using the Object-Relational Capabilities introduces the object-oriented capabilities of DB2. It explains how to extend 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.
- Chapter 11. User-defined Distinct Types describes how to create and use your own data types in applications. It explains how to use distinct types as a foundation for object-oriented extensions to the built-in data types.
- Chapter 12. Working with Complex Objects: User-Defined Structured Types describes how to create and use structured types in applications. It explains how to model objects as hierarchies of structured types, access instances of structured types as rows or columns in tables, and bind structured types into and out of your applications.
- Chapter 13. Using Large Objects (LOBs) describes how to define and use data types that can store data objects as binary or text strings of up to two gigabytes in size. It also explains how to efficiently use LOBs in a networked environment.
- Chapter 14. User-Defined Functions (UDFs) and Methods describes how to write your own extensions to SQL. It explains how to use UDFs to express the behavior of your data objects.
- Chapter 15. Writing User-Defined Functions (UDFs) and Methods describes how to write user-defined functions that extend your DB2 applications. Topics include the details of writing a user-defined function, programming considerations for user-defined functions, and several examples that show you how to exploit this important capability. In addition, this chapter describes user-defined table functions, OLE DB table functions, and OLE automation UDFs.
- Chapter 16. Using Triggers in an Active DBMS describes how to use triggers to encapsulate and enforce business rules within all of your database applications.

- Part 5. DB2 Programming Considerations contains information on special application development considerations.
  - Chapter 17. Programming in Complex Environments discusses 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 creating multi-threaded applications.
  - Chapter 18. Programming Considerations in a Partitioned Environment describes programming considerations if you are developing applications that run in a partitioned environment.
  - Chapter 19. Writing Programs for DB2 Federated Systems describes how to create applications that transparently access data from DB2 family and Oracle data sources through a federated server.

- Part 6. Language Considerations contains specific information about the programming languages that DB2 supports.
  - Chapter 20. Programming in C and C++ discusses host language specific information concerning database applications written in C and C++.
  - Chapter 21. Programming in Java discusses host language specific information concerning database applications written in Java using JDBC or SQLJ.
  - Chapter 22. Programming in Perl discusses host language specific information concerning database applications written in Perl using the DBD::DB2 database driver for the Perl Database Interface (DBI) Module.
  - Chapter 23. Programming in COBOL discusses host language specific information concerning database applications written in COBOL.
  - Chapter 24. Programming in FORTRAN discusses host language specific information concerning database applications written in FORTRAN.
  - Chapter 25. Programming in REXX discusses host language specific information concerning database applications written in REXX.
- The Appendices contain supplemental information to which you may need to refer when developing DB2 applications.
  - Appendix A. Supported SQL Statements lists the SQL statements supported by DB2 Universal Database.
  - Appendix B. Sample Programs contains information on supplied sample programs for supported host languages and describes how they work.
  - Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs contains information you can use to create stored procedures and UDFs that are compatible with previous versions of DB2 Universal Database.
  - Appendix D. Programming in a Distributed Environment Programming in a Host or AS/400 Environment describes programming considerations for DB2 Connect if you access host or AS/400 database servers in your applications in a distributed environment.
  - Appendix E. Simulating EBCDIC Binary Collation describes how to collate DB2 character strings according to an EBCDIC, or user-defined, collating sequence.
  - Appendix F. Using the DB2 Library shows you where you can get more information for the DB2 Universal Database product.

Conventions

This book uses the following conventions:

Directories and Paths

This book uses the UNIX convention for delimiting directories, for example: sqllib/samples/java. You can convert these paths to Windows 32-bit operating system and OS/2 paths by changing the / to a \ and prepending the appropriate installation drive and directory.

Italics Indicates one of the following:
- Introduction of a new term
- Variable names or values that are supplied by the user
- Reference to another source of information, for example, a book or CD-ROM
- General emphasis

UPPERCASE
Indicates one of the following:
- Abbreviations
- Database manager data types
- 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
- Java method names
- Function names
- API names

Bold Bold text emphasizes a point.

Related Publications

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

Form Number	Book Title
SE09-8001-03	*National Language Design Guide, Volume 1*
SE09-8002-03	*NLS Reference Manual, Release 4*

Chapter 2. Coding a DB2 Application

Prerequisites for Programming

This chapter presents a model of the logical parts of a DB2 application and discusses the individual strengths of the supported DB2 programming APIs. Programmers who are new to developing a DB2 application should read the entire chapter closely.

The application development process described in this book assumes that you have established the appropriate operating environment. This means that the following are properly installed and configured:

- A supported compiler or interpreter for developing your applications.

- DB2 Universal Database, either locally or remotely.
- DB2 Application Development Client.

For details on how to accomplish these tasks, refer to the *Application Building Guide* and the *Quick Beginnings* books for your operating environment.

You can *develop* applications at a server, or on any client, that has the DB2 Application Development Client (DB2 Application Development Client) installed. You can *run* applications with either the server, the DB2 Run-Time Client, or the DB2 Administrative Client. You can also develop Java JDBC programs on one of these clients, provided that you install the "Java Enablement" component when you install the client. That means you can execute any DB2 application on these clients. However, unless you also install the DB2 Application Development Client with these clients, you can only develop JDBC applications on them.

DB2 supports the C, C++, Java (SQLJ), COBOL, and FORTRAN programming languages through its precompilers. In addition, DB2 provides support for the Perl, Java (JDBC), and REXX dynamically interpreted languages. For information on the specific precompilers provided by DB2, and the languages supported on your platform, refer to the *Application Building Guide*.

Note: FORTRAN and REXX support stabilized in DB2 Version 5, and no enhancements for FORTRAN or REXX support are planned for the future.

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

DB2 Application Coding Overview

A DB2 application program consists of several parts:

1. Declaring and initializing variables
2. Connecting to the database
3. Performing one or more *transaction*s
4. Disconnecting from the database
5. 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 (SQLCA)

  Note that DB2 applications written in Java throw an `SQLException`, which you handle in a `catch` block, rather than using the SQLCA.

The body of every program contains the SQL statements that access and manage data. These statements constitute transactions. Transactions must include the following statements:
- The CONNECT statement, which establishes a connection to a database server
- One or more:
  - 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 the transaction

The end of the application program typically contains SQL statements that:
- Release the program's connection to the database server
- Clean up any resource

Declaring and Initializing Variables

To code a DB2 application, you must first declare:
- the variables that interact with the database manager
- the SQLCA, if applicable

Declaring Variables that Interact with the Database Manager

All variables that interact with the database manager must be declared in an SQL declare section. You must code an SQL declare section with the following structure:

1. the SQL statement `BEGIN DECLARE SECTION`
2. a group of one or more variable declarations
3. the SQL statement `END DECLARE SECTION`

Host program variables declared in an SQL declare section are called *host variables*. You can use host variables in *host-variable* references in SQL statements. *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 77.

Declaring Variables that Represent SQL Objects: For DB2 Version 7, the names of tables, aliases, views, and correlations have a maximum length of 128 bytes. Column names have a maximum length of 30 bytes. In DB2 Version 7, schema names have a maximum length of 30 bytes. Future releases of DB2 may increase the lengths of column names and other identifiers of SQL objects up to 128 bytes. If you declare variables that represent SQL objects with less than 128 byte lengths, future increases in SQL object identifier lengths may affect the stability of your applications. For example, if you declare the variable `char[9]`*`schema_name`* in a C++ application to hold a schema name, your application functions properly for the allowed schema names in DB2 Version 6, which have a maximum length of 8 bytes.

```
char[9] schema_name; /* holds null-delimited schema name of up to 8 bytes;
 works for DB2 Version 6, but may truncate schema names in future releases */
```

However, if you migrate the database to DB2 Version 7, which accepts schema names with a maximum length of 30 bytes, your application cannot differentiate between the schema names `LONGSCHEMA1` and `LONGSCHEMA2`. The database manager truncates the schema names to their 8-byte limit of `LONGSCHE`, and any statement in your application that depends on differentiating the schema names fails. To increase the longevity of your application, declare the schema name variable with a 128-byte length as follows:

```
char[129] schema_name; /* holds null-delimited schema name of up to 128 bytes
                          good for DB2 Version 7 and beyond */
```

To improve the future operation of your application, consider declaring all of the variables in your applications that represent SQL object names with lengths of 128 bytes. You must weigh the advantage of improved compatibility against the increased system resources that longer variables require.

To ease the use of this coding practice and increase the clarity of your C/C++ application code, consider using C macro expansion to declare the lengths of these SQL object identifiers. Since the include file `sql.h` declares SQL_MAX_IDENT to be 128, you can easily declare SQL object identifiers with the SQL_MAX_IDENT macro. For example:

```
#include <sql.h>
    char[SQL_MAX_IDENT+1] schema_name;
    char[SQL_MAX_IDENT+1] table_name;
    char[SQL_MAX_IDENT+1] employee_column;
    char[SQL_MAX_IDENT+1] manager_column;
```

For more information on C macro expansion, see "C Macro Expansion" on page 613.

Relating Host Variables to an SQL Statement: You can use host variables 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 the database manager finds a row that meets the search criteria, `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 in the CREATE TABLE definition. You must relate this data type to the host language data type defined in the *Supported SQL Data Types* section of each language-specific chapter in this document. 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++:

```
sqlint32 variable_name;
```

Java:

```
int 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 in C and C++" on page 627
- for Java, "Supported SQL Data Types in Java" on page 639
- for COBOL, "Supported SQL Data Types in COBOL" on page 695
- for FORTRAN, "Supported SQL Data Types in FORTRAN" on page 712
- for REXX, "Supported SQL Data Types in REXX" on page 726

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

After you have determined the data types, you can refer to the conversion charts in the host language chapters and code the appropriate declarations. The Declaration Generator utility (`db2dclgn`) is also available for generating the appropriate declarations for a given table in a database. For more information on `db2dclgn`, see "Declaration Generator - db2dclgn" on page 73 and refer to the *Command Reference*.

Table 4 on page 74 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. The contents of the variable determine other host variable data types and sizes at run time.

Table 4 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

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 Java applications: You do not explicitly use the SQLCA in Java. Instead, use the SQLException instance methods to get the SQLSTATE and SQLCODE values. See "SQLSTATE and SQLCODE Values in Java" on page 641 for more details.

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/ANS SQL92 and FIPS 127-2 standard.

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

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.

Note: Embedded SQL for Java (SQLJ) applications cannot use the WHENEVER statement. Use the `SQLException` methods described in "SQLSTATE and SQLCODE Values in Java" on page 641 to handle errors returned by SQL statements.

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 17 for more information on a transaction and "Diagnostic Handling and the SQLCA Structure" on page 116 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 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/ANS SQL92 or FIPS 127-2 standard, do not use the above statements or the `INCLUDE SQLCA` statement. For more information on the ISO/ANS SQL92 and FIPS 127-2 standards, see "Definition of FIPS 127-2 and ISO/ANS SQL92" on page 15. For the alternative to coding the above statements, see the following:

- For C or C++ applications, see "SQLSTATE and SQLCODE Variables in C and C++" on page 635
- For COBOL applications, "SQLSTATE and SQLCODE Variables in COBOL" on page 699
- For FORTRAN applications, "SQLSTATE and SQLCODE Variables in FORTRAN" on page 714

Using Additional Nonexecutable Statements

Generally, other *nonexecutable* SQL statements are also part of this section of the program. Both the *SQL Reference* and subsequent chapters of this manual discuss nonexecutable statements. 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 both 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 multisite update environment. In this case, only one server can be the current server. For more information on multisite updates, see "Multisite Update" on page 535.

Your program can establish a connection to a database server either:

- explicitly, using the CONNECT statement
- implicitly, connecting to the default database server
- for Java applications, through a `Connection` instance

Refer to the *SQL Reference* for a discussion of connection states and how to use the CONNECT statement. Upon initialization, the application requester establishes a default database server. If implicit connects are enabled, application processes started after initialization connect implicitly to the default database server. It is good practice to use the CONNECT statement as the first SQL statement executed by an application program. This avoids 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
- 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 multisite update 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 multisite update (see "Multisite Update" on page 535).

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*.

To ensure the consistency of data at the transaction level, the system makes sure that either *all* operations within a transaction are completed, or *none* are completed. Suppose, for example, that the program is supposed to deduct money from one account and add it to another. If you place both of these updates in a single transaction, and a system failure occurs while they are in progress, then when you restart the system, the database manager automatically restores the data to the state it was in before the transaction began. If a program error occurs, the database manager restores all changes made by the statement in error. The database manager will not undo work performed in the transaction prior to execution of the statement in error, 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

multisite update. For information on these topics, see "Remote Unit of Work" on page 535 and "Multisite Update" on page 535.

Beginning a Transaction

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

In contrast, the following six statements 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 a program contains an executable SQL statement after a transaction ends, it automatically starts a new transaction.

Ending a Transaction

To end a transaction, you can use either:

- The COMMIT statement to save its changes
- 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 performed during the current transaction visible to other processes.

You should commit changes 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.

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

Your application programs should explicitly end any transactions prior to terminating. If you do not end transactions explicitly, DB2 automatically commits all the changes made during the program's pending transaction when the program ends successfully, except on Windows 32-bit operating systems. DB2 rolls back the changes under the following conditions:

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

On Windows 32-bit operating systems, if you do not explicitly commit the transaction, the database manager always rolls back the changes.

For more information about program termination, see "Ending the Program" and "Diagnostic Handling and the SQLCA Structure" on page 116.

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

The ROLLBACK statement has no effect on the contents of host variables.

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.

In the event of a severe error, you will receive a message indicating that you cannot issue a ROLLBACK statement. Do not issue a ROLLBACK statement if a severe error occurs such as the loss of communications between the client and server applications, or if the database gets corrupted. After a severe error, the only statement you can issue 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 on 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" 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 DB2 issues a COMMIT or ROLLBACK depends on 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 543)

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 32-bit 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 543. If your application uses these DB2 APIs, DB2 implicitly rolls back the transaction regardless of whether your application terminates normally or abnormally. Unless you explicitly commit the transaction using the COMMIT statement, DB2 rolls back the transaction.

Application Pseudocode Framework

Pseudocode Framework for Coding Programs summarizes the general framework for a DB2 application program in pseudocode format. You must, of course, tailor this framework to suit your own program.

```
Start Program
EXEC SQL BEGIN DECLARE SECTION                                  |
  DECLARE USERID FIXED CHARACTER (8)                            |
  DECLARE PW FIXED CHARACTER (8)                                |
                                                                | Application
  (other host variable declarations)                            | Setup
                                                                |
EXEC SQL END DECLARE SECTION                                    |
EXEC SQL INCLUDE SQLCA                                          |
EXEC SQL WHENEVER SQLERROR GOTO ERRCHK                          |

  (program logic)

EXEC SQL CONNECT TO database A USER :userid USING :pw           |
EXEC SQL SELECT ...                                             |
EXEC SQL INSERT ...                                             | First Unit
  (more SQL statements)                                         | of Work
EXEC SQL COMMIT                                                 |

  (more program logic)
```

```
EXEC SQL CONNECT TO database B USER :userid USING :pw
EXEC SQL SELECT ...
EXEC SQL DELETE ...                                        Second Unit
  (more SQL statements)                                    of Work
EXEC SQL COMMIT

  (more program logic)

EXEC SQL CONNECT TO database A
EXEC SQL SELECT ...
EXEC SQL DELETE ...                                        Third Unit
  (more SQL statements)                                    of Work
EXEC SQL COMMIT

  (more program logic)

EXEC SQL CONNECT RESET
ERRCHK
                                                           Application
  (check error information in SQLCA)                       Cleanup

End Program
```

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, you must make the most fundamental design decision: *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 enforce some business rules in your 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, including embedded SQLJ for Java (SQLJ)
  - DB2 Call Level Interface (DB2 CLI), Open Database Connectivity (ODBC), and Java Database Connectivity (JDBC)
  - Microsoft Specifications
  - Perl DBI
  - 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 this list mentions some capabilities more than once, such as triggers. This reflects the flexibility of these capabilities to address more than one design criteria.

Your 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 you can change data or data logic 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 need to process validation errors on data entry in a specific order. In general, you should do these types of data validation 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 using either stored procedures, UDFs, or a combination of both.

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 you may choose how to integrate the SQL into your application. You can choose from the following interfaces and their supported languages:

Embedded SQL
C/C++, COBOL, FORTRAN, Java (SQLJ), REXX

DB2 CLI and ODBC
C/C++, Java (JDBC)

Microsoft Specifications, including ADO, RDO, and OLE DB
Visual Basic, Visual C++

Perl DBI
Perl

Query Products
Lotus Approach, IBM Query Management Facility

Embedded SQL

Embedded SQL has the advantage that it can consist of either static or dynamic SQL or a mixture of both types. If the content and format of your SQL statements will be *frozen* when your application is in use, you should consider using embedded static SQL in your application. With static SQL, the person who executes the application temporarily inherit the privileges of the user that bound the application to the database. Unless you bind the application with the DYNAMICRULES BIND option, dynamic SQL uses the privileges of the person who executes the application. In general, you should use embedded dynamic SQL where the executable statements are determined at run time. This creates a more secure application program that can handle a greater variety of input.

Note: Embedded SQL for Java (SQLJ) applications can only embed static SQL statements. However, you can use JDBC to make dynamic SQL calls in SQLJ applications.

You must precompile embedded SQL applications to convert the SQL statements into host language commands before using your programming language compiler. In addition, you must bind the SQL in the application to the database for the application to run.

For additional information on using embedded SQL, refer to “Chapter 4. Writing Static SQL Programs” on page 61.

REXX Considerations: 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, a dynamic SQL handler processes all SQL statements. By combining REXX with these callable APIs, you have access to most of the database manager capabilities. Although REXX does not directly support some APIs using embedded SQL, 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 25. Programming in REXX" on page 717.

DB2 Call Level Interface (DB2 CLI) and Open Database Connectivity (ODBC)

The DB2 Call Level Interface (DB2 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 precompiling or binding.

DB2 CLI is based on the Microsoft™ Open Database Connectivity (ODBC) specification, and the X/Open® specifications. IBM chose these specifications to follow industry standards, and to provide a shorter learning curve for DB2 application programmers who are familiar with either of these database interfaces.

For more information on the ODBC support in DB2, see the *CLI Guide and Reference*.

JDBC

DB2's Java support includes JDBC, a vendor-neutral dynamic SQL interface that provides data access to your application through standardized Java methods. JDBC is similar to DB2 CLI in that you do not have to precompile or bind a JDBC program. As a vendor-neutral standard, JDBC applications offer increased portability.

An application written using JDBC uses only dynamic SQL. The JDBC interface imposes additional processing overhead.

For additional information on JDBC, refer to "JDBC Programming" on page 644.

Microsoft Specifications

You can write database applications that conform to the ActiveX Data Object (ADO) in Microsoft Visual Basic™ or Visual C++™. ADO applications use the OLE DB Bridge. You can write database applications that conform to the Remote Data Object (RDO) specifications in Visual Basic. You can also define OLE DB table functions that return data from OLE DB providers. For more information on OLE DB table functions, see "OLE DB Table Functions" on page 431.

This book does not attempt to provide a tutorial on writing applications that conform to the ADO and RDO specifications. For full samples of DB2 applications that use the ADO and RDO specifications, refer to the following directories:

- For samples written in Visual Basic, refer to `sqllib\samples\VB`
- For samples written in Visual C++, refer to `sqllib\samples\VC`
- For samples that use the RDO specification, refer to `sqllib\samples\RDO`
- For samples that use the Microsoft Transaction Server™, refer to `sqllib\samples\MTS`

Perl DBI

DB2 supports the Perl Database Interface (DBI) specification for data access through the DBD::DB2 driver. For more information on creating appliations with the Perl DBI that access DB2 databases, see "Chapter 22. Programming in Perl" on page 675. The DB2 Universal Database Perl DBI Web site at `http://www.ibm.com/software/data/db2/perl/` contains the latest DBD::DB2 driver and information on the support available for your platform.

Query Products

Query products including IBM Query Management Facility (QMF) and Lotus Notes 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. This book does not provide further information on query products.

Data Value Control

One traditional area of application logic is validating and protecting data integrity by controlling the values allowed in the database. Applications have logic that specifically checks data values as they are entered for validity. (For example, checking that the department number is a valid number and that it

refers to an existing department.) There are several different ways of providing these same capabilities in DB2, but from within the database.

Data Types

The database stores every data element in a column of a table, and defines each column with a data type. This data type places 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, the database does not compare an integer 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*.

Unique Constraints

Unique constraints prevent occurrences of duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, you can define a unique constraint on the DEPTNO column in the DEPARTMENT table to ensure that the same department number is not given to two departments.

Use unique constraints if you need to enforce a uniqueness rule for all applications that use the data in a table. For additional information on unique constraints, refer to the *SQL Reference*.

Table Check Constraints

You can use a table check constraint to define restrictions, beyond those of the data type, on the values that are allowed for a column in the table. Table check constraints take the form of range checks or checks against other values in the same row of the same table.

If the rule applies for all applications that use the data, use a table check constraint to enforce your restriction on the data allowed in the table. Table check constraints make the restriction generally applicable and easier to maintain.

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

Referential Integrity Constraints

Use referential integrity (RI) constraints if you must maintain value-based relationships for all applications that use the data. For example, you can use an RI constraint to ensure that the value of a DEPTNO column in an

EMPLOYEE table matches a value in the DEPARTMENT table. This constraint prevents inserts, updates or deletes that would otherwise result in missing DEPARTMENT information. By centralizing your rules in the database, RI constraints make the rules generally applicable and easier to maintain.

See "Data Relationship Control" for further uses of RI constraints.

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

Views with Check Option

If your application cannot define the desired rules as table check constraints, or the rules 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 retrieval of data to the set that is valid for your application. Additionally, if you can update the view, 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

When you write your application logic in a programming language, you also declare variables to provide some of the same restrictions on data that are described above. In addition, you can choose to write code to enforce rules in the application instead of the database. Place the logic in the application server when:

- The rules are not generally applicable, except in the case of views noted in "Views with Check Option"
- You do not have control over the definitions of the data in the database
- 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 you add a new department, then you need to create a new account code. 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. Use the CREATE TABLE or ALTER TABLE statements to define the behavior of operations that affect the related primary key, such as DELETE and UPDATE.

RI constraints enforce your rules on the data across one or more tables. If the rules apply for all applications that use the data, then RI constraints centralize the rules in the database. This makes the rules generally applicable and easier to maintain.

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

Triggers

You can use triggers before or after an update to support logic that can also be performed in an application. If the rules or operations supported by the triggers apply for all applications that use the data, then triggers centralize the rules or operations in the database, making it generally applicable and easier to maintain.

For additional information on triggers, see "Chapter 16. Using Triggers in an Active DBMS" on page 483 and refer to the *SQL Reference*.

Using Triggers Before an Update: 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 After an Update: Triggers that run after an update, insert or delete can be used in several ways:

- Triggers can update, insert, or delete data in the same or other tables. This is useful to maintain relationships between data or to keep audit trail information.
- Triggers can check data against values of data in the rest of the table or in other tables. This is useful when you cannot use RI constraints or check constraints because of references to data from other rows from this or other tables.
- Triggers can use user-defined functions to activate non-database operations. This is useful, for example, for issuing alerts or updating information outside the database.

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 you cannot generally apply the rules to the database. 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 application logic can handle the rules or operations more efficiently.

Application Logic at the Server

A final aspect of application design for which DB2 offers additional capability is running some of your application logic at the database server. Usually you will choose this design to improve performance, but you may also run application logic at the server to support common functions.

This aspect of application logic is described further in the following section:

- Stored Procedures
- User-Defined Functions
- Triggers

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 7. Stored Procedures" on page 193.

User-Defined Functions

You can write a user-defined function (UDF) for use in performing operations within an SQL statement to return:

- A single scalar value (scalar function)
- A table from a non-DB2 data source, for example, an ASCII file or a Web page (table function)

A UDF cannot contain SQL statements. UDFs are useful for tasks like 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).

For additional information on writing user-defined functions, refer to "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393.

Triggers

In "Triggers" on page 28, 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 16. Using Triggers in an Active DBMS" on page 483.

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ is a collection of management tools and wizards that plug into the Visual C++ component of Visual Studio IDE. The tools and wizards automate and simplify the various tasks involved in developing applications for DB2 using embedded SQL.

You can use the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ to develop, package, and deploy:

- Stored procedures written in C/C++ for DB2 Universal Database on Windows 32-bit operating systems
- Windows 32-bit C/C++ embedded SQL client applications that access DB2 Universal Database servers
- Windows 32-bit C/C++ client applications that invoke stored procedures using C/C++ function call wrappers

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ allows you to focus on the design and logic of your DB2 applications rather than the actual building and deployment of it.

Some of the tasks performed by the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ include:

- Creating a new embedded SQL module
- Inserting SQL statements into an embedded SQL module using SQL Assist
- Adding imported stored procedures
- Creating an exported stored procedure
- Packaging the DB2 Project
- Deploying the DB2 project from within Visual C++

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ is presented in the form of a toolbar. The toolbar buttons include:

DB2 Project Properties
Manages the project properties (development database and code-generation options)

New DB2 Object
Adds a new embedded SQL module, imported stored procedure, or exported stored procedure

DB2 Embedded SQL Modules
Manages the list of embedded SQL modules and their precompiler options

DB2 Imported Stored Procedures
Manages the list of imported stored procedures

DB2 Exported Stored Procedures
Manages the list of exported stored procedures

Package DB2 Project
Packages the DB2 external project files

Deploy DB2 Project
Deploys the packaged DB2 external project files

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ also has the following three hidden buttons that can be made visible using the standard Visual C++ tools customization options:

New DB2 Embedded SQL Module
Adds a new C/C++ embedded SQL module

New DB2 Imported Stored Procedure
Imports a new database stored procedure

New DB2 Exported Stored Procedure
Exports a new database stored procedure

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ can automatically generate the following code elements:

- Skeletal embedded SQL module files with optional sample SQL statements
- Standard database connect and disconnect embedded SQL functions
- Imported stored procedure call wrapper functions
- Exported stored procedure function templates
- Exported stored procedure data definition language (DDL) files

Terminology associated with the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++:

IDE project
The standard Visual C++ project

DB2 project
The collection of DB2 project objects that are inserted into the IDE project. DB2 project objects can be inserted into any Visual C++ project. The DB2 project allows you to manage the various DB2 objects such as embedded SQL modules, imported stored procedures, and exported stored procedures. You can add, delete, and modify these objects and their properties.

module
A C/C++ source code file that might contain SQL statements.

development database
The database that is used to compile embedded SQL modules. The development database is also used to look up the list of importable database stored procedure definitions.

embedded SQL module
A C/C++ source code file that contains embedded static or dynamic SQL.

imported stored procedure
A stored procedure, already defined in the database, that the project invokes.

exported stored procedure
A database stored procedure that is built and defined by the project.

Activating the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

To activate the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++, perform the following steps:

Step 1. Start and stop Visual C++ at least once with your current login ID. The first time you run Visual C++, a profile is created for your user ID, and that is what gets updated by the `db2vccmd` command. If you have not started it once, and you try to run `db2vccmd`, you may see errors like the following:

```
"Registering DB2 Project add-in ...Failed! (rc = 2)"
```

Step 2. Register the add-in, if you have not already done so, by entering:

```
db2vccmd register
```

on the command line.

Step 3. Select **Tools —> Customize**. The Customize notebook opens.

Step 4. Select the **Add-ins and Macro Files** tab. The Add-ins and Macro Files page opens.

Step 5. Select the **IBM DB2 Project Add-In** check box.

Step 6. Click **OK**. A floating toolbar will be created.

Note: If the toolbar is accidentally closed, you can either deactivate then reactivate the add-in or use the Microsoft Visual C++ standard customization options to redisplay the toolbar.

Activating the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++

The DB2 Tools Add-In is a toolbar that enables the launch of some of the DB2 administration and development tools from within the Visual C++ integrated development environment.

To activate the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++, perform the following steps:

Step 1. Start and stop Visual C++ at least once with your current login ID. The first time you run Visual C++, a profile is created for your user ID, and that is what gets updated by the `db2vccmd` command. If you have not started it once, and you try to run `db2vccmd`, you may see errors like the following:

```
"Registering DB2 Project add-in ...Failed! (rc = 2)"
```

Step 2. Register the add-in, if you have not already done so, by entering:

```
db2vccmd register
```

on the command line.

Step 3. Select **Tools —> Customize**. The Customize notebook opens.

Step 4. Select the Add-ins and Macro Files tab.

Step 5. Select the **IBM DB2 Tools Add-In** check box.

Step 6. Click **OK**. A floating toolbar will be created.

Note: If the toolbar is accidentally closed, you can either deactivate then reactivate the add-in or use the Visual C++ standard customization options to redisplay the toolbar.

For more information on the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++, refer to:

- The online help for the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++.
- http://www.ibm.com/software/data/db2/udb/ide/index.html.

Supported SQL Statements

The SQL language provides for data definition, retrieval, update, and control operations from within an application. Table 38 on page 737 shows the SQL statements supported by the DB2 product and whether the statement is supported dynamically, through the CLP, or through the DB2 CLI. You can

use Table 38 on page 737 as a quick reference aid. For a complete discussion of all the statements, including their syntax, refer to 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. For more information about the different privileges, refer to the *Administration Guide: Planning*.

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 the *Administrative API Reference*.

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

When you design your application, consider the privileges your users will need to run the application. The privileges required by your users depend on:

- whether your application uses dynamic SQL, including JDBC and DB2 CLI, or static SQL
- which APIs the application uses

Dynamic SQL

To use dynamic SQL in a package bound with DYNAMICRULES RUN (default), the person that runs a dynamic SQL 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.

If you bind the application with the DYNAMICRULES BIND option, DB2 associates your authorization ID with the application packages. This allows any user that runs the application to inherit the privileges associated your authorization ID.

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.

When you bind a dynamic SQL package with the DYNAMICRULES BIND option, the user that runs the application only needs the EXECUTE privilege on the package. To bind a dynamic SQL application with the DYNAMICRULES BIND option, you must have the privileges necessary to perform all the dynamic and static SQL statements in the application. If you have SYSADM or DBADM authority and bind packages with DYNAMICRULES BIND, consider using the OWNER BIND option to designate a different authorization ID. OWNER BIND prevents the package from automatically inheriting SYSADM or DBADM privileges on dynamic SQL statements. For more information on DYNAMICRULES BIND and OWNER BIND, refer to the BIND command in the *Command Reference*.

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.

Unless you specify the VALIDATE RUN option when binding the application, the authorization ID you use to bind the application must have the privileges necessary to perform all the statements in the application. If VALIDATE RUN was specified at BIND time, all authorization failures for any static SQL within this package will not cause the BIND to fail and those statements will be revalidated at run time. The person binding the application must always have 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.

These properties of static SQL give you 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 the use of privileges, 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 *Administrative 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 issues 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 SQL or DB2 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, refer to the examples in the *Administrative API Reference*.

You can use the DB2 APIs to:

- Manipulate the database manager environment, which includes cataloging and uncataloging 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.

The database manager includes 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 *Administrative 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, use test data 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 used to test an application 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 you must create a test database, write a small server application that calls the CREATE DATABASE API, or use the command line processor. Refer to the *Command Reference* for information about the command line processor, or the *Administrative 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. You could record the type of accesses as shown in Table 1 on page 38.

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

Use any of the following methods to insert data into a table:

- 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 inserts large amounts of new or existing data from a defined source.
- 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, refer to the *SQL Reference*. For information about the IMPORT, LOAD, and RESTORE utilities, refer to the *Administration Guide*.

The following SQL statements demonstrate a technique you can use to populate your tables with randomly generated test data. Suppose the table EMP 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 ) 1

-- Now, use the generated records in DT to create other columns
-- of the employee record.
SELECT ENO, 2
    TRANSLATE(CHAR(INTEGER(RAND()*1000000)), 3
              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, 4
    INTEGER(10000+RAND()*200000) AS SALARY 5
  FROM DT;

SELECT * FROM EMP;
```

The following is an explanation of the above statement:

1. The first part of the 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 sample programs that are helpful in generating random test data, 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 15. Writing User-Defined Functions (UDFs) and Methods" on page 393 and "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373.

Running, Testing and Debugging Your Programs

The *Application Building Guide* 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" on page 41. These include using the command line processor, the Explain facility, analyzing the system catalog views for 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 Universal Database for 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 49, towards the end of the section.

- Make full use of the error-handling APIs. For example, you can use error-handling APIs to print all messages during the testing phase. For more information about error-handling APIs, see the *Administrative API Reference*.

Prototyping Your SQL Statements

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

- Use the Command Center or the command line processor (CLP) to test many SQL statements before you attempt 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 syntax for some SQL statements in order to use host variables in your embedded SQL program. Host variables are used to store data that is output to your screen. In addition, some embedded SQL statements (such as BEGIN DECLARE SECTION) are not supported by the Command Center or CLP as they are not relevant to that environment. See Table 38 on page 737 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, to save retyping the statement.

  For information about the command line processor, refer to the *Command Reference*. For information about the Command Center, refer to the *Administration Guide*.

- 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, refer to the *Administration Guide: Implementation*.

- Use the system catalog views to easily retrieve information about existing databases. The database manager creates and maintains the system catalog tables on which the views are based during normal operation as databases are created, altered, and updated. These views contain data about each database, including 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. You may change some columns in these views 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, refer to the appendix in the *SQL Reference*. For information about system catalog statistics and which ones you can change, refer to the *Administration Guide: Implementation*.

Part 2. Embedding SQL in Applications

Chapter 3. Embedded SQL Overview

Embedding SQL Statements in a Host Language

You can write applications 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.

Table 2 shows an SQL statement embedded in a host language application. In the example, the application checks the SQLCODE field of the SQLCA structure to determine whether the update was successful.

Table 2. Embedding SQL Statements in a Host Language

Language	Sample Source Code

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

Java (SQLJ)	`try {`
	`  #sql { UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' };`
	`}`
	`catch (SQLException e) {`
	`  println( "Update Error:  SQLCODE = " + e.getErrorCode() );`
	`}`

COBOL	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`
	`IF SQLCODE LESS THAN 0`
	`  DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.`

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

SQL statements placed in an application are not specific to the host language. The database manager provides a way to convert the SQL syntax for processing by the host language.

For the C, C++, COBOL or FORTRAN languages, this conversion is handled by the DB2 precompiler. The DB2 precompiler is invoked using the `PREP` command. The precompiler converts embedded SQL statements directly into DB2 run-time services API calls.

For the Java language, the SQLJ translator converts SQLJ clauses into JDBC statements. The SQLJ translator is invoked with the `SQLJ` 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 in C and C++" on page 599
- for Java (SQLJ), "Embedding SQL Statements in Java" on page 654
- for COBOL, "Embedding SQL Statements in COBOL" on page 683
- for FORTRAN, "Embedding SQL Statements in FORTRAN" on page 705
- for REXX, "Embedding SQL Statements in REXX" on page 719

Table 3 shows how to use delimiters and comments to create valid embedded SQL statements in the supported compiled host languages.

Table 3. Embedding SQL Statements in a Host Language

Language	Sample Source Code

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

SQLJ	`/* Only Java comments allowed here */`
	`#sql c1 = {`
	`  -- SQL comments or`
	`  /* Java comments or */`
	`  // Java comments allowed here`
	`  SELECT name FROM employee`
	`};`
	`/* Only Java comments allowed here */`

Table 3. Embedding SQL Statements in a Host Language (continued)

Language	Sample Source Code

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

FORTRAN	`C Only FORTRAN comments are allowed here`
	`EXEC SQL`
	`+ -- SQL comments, and`
	`C full-line FORTRAN comment are allowed here`
	`+ DECLARE C1 CURSOR FOR sname`
	`I=7 ! End of line FORTRAN comments allowed here`
	`C Only FORTRAN comments are allowed here`

Creating and Preparing the Source Files

You can create the source code 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 39 on page 745 to find out the required file extension for the host language you are using.

Note: Not all platforms support all host languages. See the *Application Building Guide* for specific information.

For this discussion, assume that you have already written the source code.

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

Simply stated, precompiling converts embedded SQL statements into DB2 run-time API calls that a host compiler can process, and creates a bind file. The bind file contains information on the SQL statements in the application program. The `BIND` command creates a *package* in the database. 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, you must create a package for each database.

Figure 1 on page 48 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.

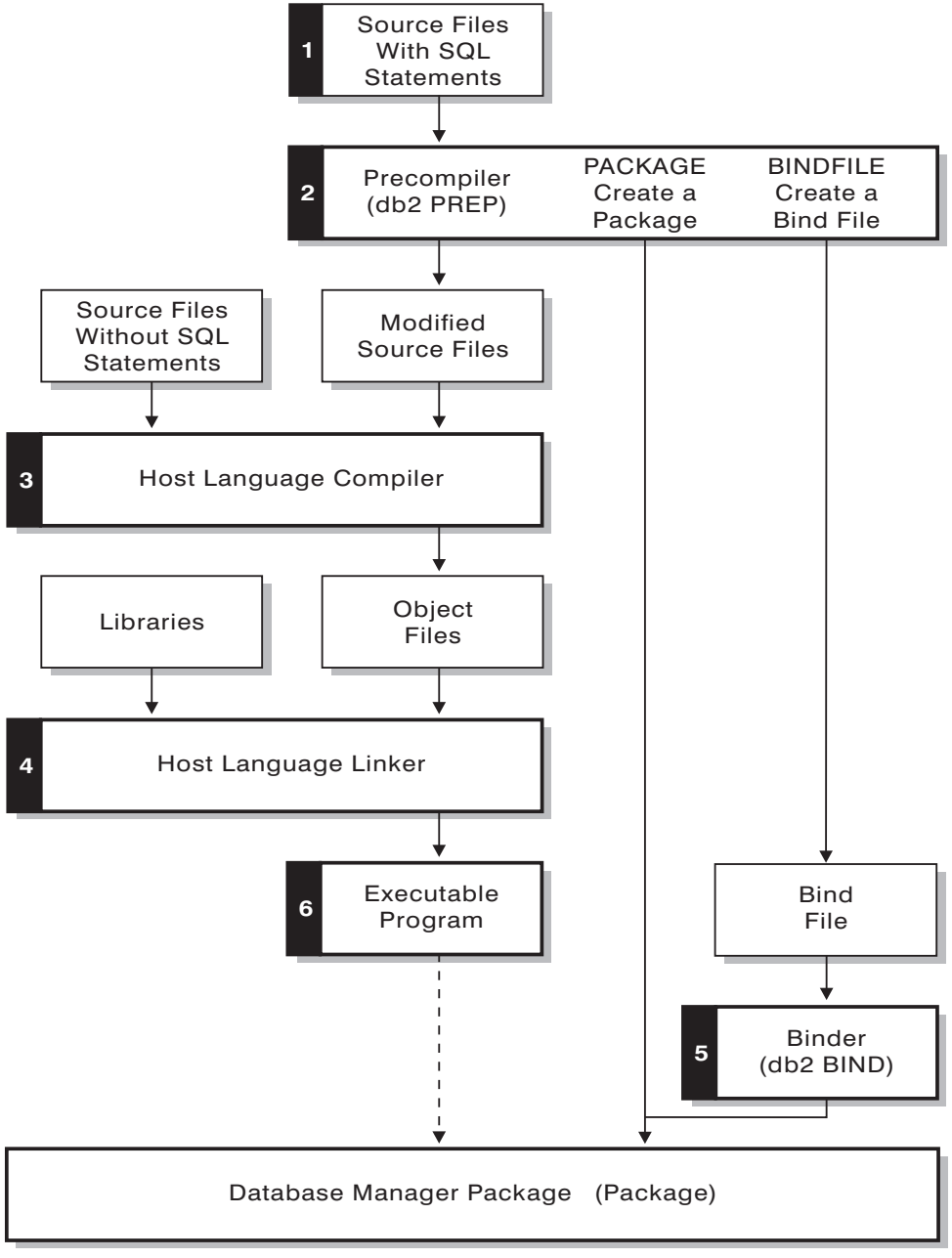


Figure 1. Preparing Programs Written in Compiled Host Languages

Creating Packages for Embedded SQL

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 as shown in Figure 1 on page 48:

- Precompiling (step 2), to convert embedded SQL source statements into a form the database manager can use,
- Compiling and Linking (steps 3 and 4), to create the required object modules, and,
- Binding (step 5), 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.

To create the packages needed by SQLJ applications, you need to use both the `SQLJ` translator and `db2profc` command. For more information on using the `SQLJ` translator, see "SQLJ Programming" on page 651.

Precompiling

After you create the source files, you must precompile each host language file containing SQL statements with the `PREP` command for host language source files. 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 you precompile application programs at the client workstation and the precompiler generates modified source and messages on the client, the precompiler uses the server connection to perform some of the validation.

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.

A typical example of using the precompiler follows. To precompile a C embedded SQL source file called *filename.sqc*, you can issue the following command to create a C source file with the default name `filename.c` and a bind file with the default name `filename.bnd`:

```
DB2 PREP filename.sqc BINDFILE
```

For detailed information on precompiler syntax and options, see the *Command Reference*.

The precompiler generates up to four types of output:

- Modified source
- Package
- Bind file
- Message file

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 you use the PACKAGE option (the default), or do not specify any of the BINDFILE, SYNTAX, or SQLFLAG options, the package is stored in the connected database. The package contains all the information required to execute the static SQL statements of a particular source file against this database only. Unless you specify a different name with the PACKAGE USING option, the precompiler forms the package name from the first 8 characters of the source file name.

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, you cannot precompile a SELECT statement unless the table it references exists in the database.

Bind File

If you use the BINDFILE option, 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 refer to the *Command Reference*.

Message File If you use the MESSAGES option, the precompiler redirects messages to the indicated file. These messages include warnings and error messages that describe 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 you do not use the MESSAGES option, precompilation messages are written to the standard output.

Source File Requirements

You must always precompile a source file against a specific database, even if eventually you do not use the database with the application. In practice, you can use a test database for development, and after you fully test the application, you can bind its bind file to one or more production databases. See "Advantages of Deferred Binding" on page 56 for other ways to use this feature.

If your application uses a code page that 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 515.

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 refer to 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 host or AS/400 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 Universal Database for 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 Universal Database for 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 Universal Database for OS/390 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 does not terminate preprocessing abnormally.

For details about the PREP command, refer to 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*.

Refer to the *Application Building Guide* or other programming documentation for your operating platform for any exceptions to the default compile options. Refer to your compiler's documentation for a complete description of available compile options.

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

- On OS/2 and Windows 32-bit 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 32-bit operating systems, the DLLs are loaded directly by the application and not by the DB2 database manager. On Windows 32-bit operating systems, the database manager can load DLLs. Stored procedures are normally built as DLLs or shared libraries. For information on using stored procedures, see "Chapter 7. Stored Procedures" on page 193.

For information on creating executable files on other platforms supported by DB2, refer to the *Application Building Guide*.

To create the executable file, link 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. Refer to the *Application Building Guide* 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.

A typical example of using the `BIND` command follows. To bind a bind file named *filename.bnd* to the database, you can issue the following command:

```
DB2 BIND filename.bnd
```

For detailed information on `BIND` command syntax and options, refer to the *Command Reference*.

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. To make subsequent changes, 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 58 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 values of a number of special registers determine the statement compilation environment:

- The CURRENT QUERY OPTIMIZATION special register determines which optimization class is used.
- The CURRENT FUNCTION PATH special register determines the function path used for UDF and UDT resolution.
- The CURRENT EXPLAIN SNAPSHOT register determines whether explain snapshot information is captured.
- 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 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 *user_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 default qualifier is 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 uses a different code page from your database code page, you may need to consider which code page to use when binding. See "Conversion Between Different Code Pages" on page 515.

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

You can use bind options to control certain operations that occur during binding, as in the following examples:

- The QUERYOPT bind option takes advantage of a specific optimization class when binding.
- The EXPLSNAP bind option stores Explain Snapshot information for eligible SQL statements in the Explain tables.
- The FUNCPATH bind option properly resolves 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 connected to the database hold 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 server using DB2 Connect, you can use the `BIND` options available for that server. For details on the `BIND` command and its options, refer to the *Command Reference*.

Bind files are not backward compatible with previous versions of DB2 Universal Database. In mixed-level environments, DB2 can only use the functions available to the lowest level of the database environment. For example, if a V5.2 client connects to a V5.0 server, the client will only be able to use V5.0 functions. As bind files express the functionality of the database, they are subject to the mixed-level restriction.

If you need to rebind higher-level bind files on lower-level systems, you can:

- Use a lower-level DB2 Application Development Client to connect to the higher-level server and create bind files which can be shipped and bound to the lower-level DB2 Universal Database environment.
- Use a higher-level DB2 client in the lower-level production environment to bind the higher-level bind files that were created in the test environment. The higher-level client passes only the options that apply to the lower-level server.

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. You can design the application to bind itself to a database only when the application calls the task requiring SQL statements, and only if an associated package does not already exist.

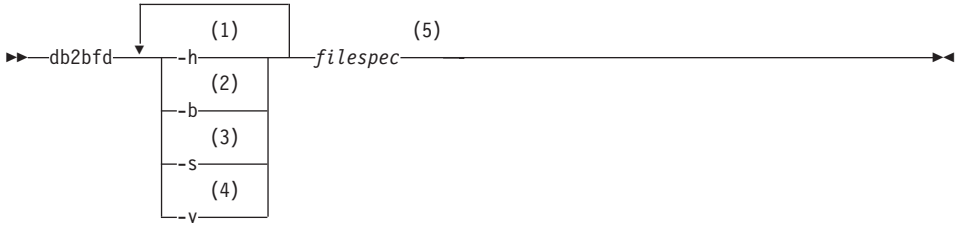
Another advantage of the deferred binding method is that it lets you create packages without providing source code to end users. You can ship the associated bind files with the application.

DB2 Bind File Description Utility - db2bfd

With the DB2 Bind File Description (`db2bfd`) utility, you can easily display the contents of a 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` utility is located in the `bin` subdirectory of the `sqllib` directory of the instance.

Its syntax is:



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.

For more information on `db2bfd`, refer to the *Command Reference*.

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 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 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, static SQL is faster 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 128.

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 53.

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, refer to the *Administration Guide: Planning*.

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. 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, refer to the *Command Reference*.

Chapter 4. Writing Static SQL Programs

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, refer to 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 the following supported languages:

C	static.sqc
Java	Static.sqlj
COBOL	static.sqb

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 71. 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 81 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 71.

How the Static 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 71.
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.** 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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

   Java Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.

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

   See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

6. **Disconnect from database.** The program disconnects from the database by executing the CONNECT RESET statement. Note that SQLJ programs automatically close the database connection when the program returns.

C Example: STATIC.SQC

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

EXEC SQL INCLUDE SQLCA;  1

int main(int argc, char *argv[])
{   int  rc = 0;

    char dbAlias[15] ;
    char user[15] ;
    char pswd[15] ;

    EXEC SQL BEGIN DECLARE SECTION;  2
        char firstname[13];
    EXEC SQL END DECLARE SECTION;

    /* checks the command line arguments */
    rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );  3
    if ( rc != 0 ) return( rc ) ;

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

    /* initialize the embedded application */
    rc = EmbAppInit( dbAlias, user, pswd);
    if ( rc != 0 ) return( rc ) ;

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

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

    /* terminate the embedded application */
    rc = EmbAppTerm( dbAlias);
    return( rc ) ;
}
/* end of program : STATIC.SQC */
```

Java Example: Static.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Static
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
      }
      catch (Exception e)
      {   System.out.println ("\n  Error loading DB2 Driver...\n");
        System.out.println (e);
        System.exit(1);
      }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Static Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                 3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Static [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      String firstname = null;

      #sql { SELECT FIRSTNME INTO :firstname
           FROM employee
           WHERE LASTNAME = 'JOHNSON' } ;    4


      System.out.println ("First name = " + firstname);
    }

    catch( Exception e )  5
```

```
    {
      System.out.println (e);
    }
  }
}
```

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.
```

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 20. Programming in C and C++" on page 593
- "Chapter 21. Programming in Java" on page 637
- "Chapter 23. Programming in COBOL" on page 679
- "Chapter 24. Programming in FORTRAN" on page 701
- "Chapter 25. Programming in REXX" on page 717.

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 81,
- "Selecting Multiple Rows Using a Cursor" on page 81,
- "Example: Cursor Program" on page 84.

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.

Note: Java JDBC and SQLJ programs do not use declare sections. Host variables in Java follow the normal Java variable declaration syntax.

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

- "Chapter 20. Programming in C and C++" on page 593
- "Chapter 21. Programming in Java" on page 637
- "Chapter 23. Programming in COBOL" on page 679
- "Chapter 24. Programming in FORTRAN" on page 701
- "Chapter 25. Programming in REXX" on page 717.

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

For further information on declaring host variables, see:

- "Declaration Generator - db2dclgn" to generate host variable declaration source code automatically with the db2dclgn tool
- Table 4 on page 74 for examples of how host variables appear in source code
- Table 5 on page 75 for examples of how to reference host variables in the supported host languages
- "Naming Host Variables in REXX" on page 721 and "Referencing Host Variables in REXX" on page 721 for information on naming and referencing host variables in REXX

Declaration Generator - db2dclgn

The Declaration Generator speeds application development by generating declarations for a given table in a database. It creates embedded SQL declaration source files which you can easily insert into your applications. db2dclgn supports the C/C++, Java, COBOL, and FORTRAN languages.

To generate declaration files, enter the db2dclgn command in the following format:

```
db2dclgn -d database-name -t table-name [options]
```

For example, to generate the declarations for the STAFF table in the SAMPLE database in C in the output file `staff.h`, issue the following command:

```
db2dclgn -d sample -t staff -l C
```

The resulting `staff.h` file contains:

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[5];
  short years;
  double salary;
  double comm;
} staff;
```

For detailed information on `db2dclgn`, refer to the *Command Reference*.

Table 4. Declaring Host Variables

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;`

Java	`// Note that Java host variable declarations follow`
	`// normal Java variable declaration rules, and have`
	`// no equivalent of a DECLARE SECTION`
	`  short  dept=38, age=26;`
	`  double  salary;`
	`  char  CH;`
	`  String  name1[9], NAME2[9];`
	`  /* Java comment */`
	`  short  nul_ind;`

Table 4. Declaring Host Variables (continued)

Language	Example Source Code

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`

Table 5. Referencing Host Variables

Language	Example Source Code
C/C++	`EXEC SQL FETCH C1 INTO :cm;` `printf( "Commission = %f\n", cm );`
JAVA (SQLJ)	`#SQL { FETCH :c1 INTO :cm };` `System.out.println("Commission = " + cm);`
COBOL	`EXEC SQL FETCH C1 INTO :cm END-EXEC` `DISPLAY 'Commission = ' cm`
FORTRAN	`EXEC SQL FETCH C1 INTO :cm` `WRITE(*,*) 'Commission = ', cm`

Using Indicator Variables

Applications written in languages other than Java must prepare for receiving null values by associating an *indicator variable* with any host variable that can receive a null. Java applications compare the value of the host variable with Java `null` to determine whether the received value is 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 SQL data 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.

Indicator Variables 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" );
```

Java (SQLJ)

```
#SQL { FETCH :c1 INTO :cm };
if ( cm == null )
   System.out.println( "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
```

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. The database manager does not change the value of the host variable in this case.

Note: If the database configuration parameter DFT_SQLMATHWARN 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.

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.

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, refer to 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, refer to 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.

BIGINT
64-bit signed integer.

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 32672 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 16336 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 515.

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 in C and C++" on page 627
- for Java, "Supported SQL Data Types in Java" on page 639
- for COBOL, "Supported SQL Data Types in COBOL" on page 695
- for FORTRAN, "Supported SQL Data Types in FORTRAN" on page 712
- for REXX, "Supported SQL Data Types in REXX" on page 726.

For more information about SQL data types, the rules of assignments and comparisons, and data conversion and conversion errors, refer to 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 66. 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 84 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.

Declare Cursor Statement 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;
```

Java (SQLJ)

```
#sql iterator cursor1(host_var data type);
#sql cursor1 = { 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
```

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.

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.

WITH RELEASE Option: When an application closes a cursor using the WITH RELEASE option, DB2 attempts to release all READ locks that the cursor still holds. The cursor will only continue to hold WRITE locks. If the application closes the cursor without using the RELEASE option, the READ and WRITE locks will be released when the unit of work completes.

Example: Cursor Program

This sample program shows the SQL statements that define and use a cursor. The cursor is processed using static SQL. The sample is available in the following programming languages:

C	cursor.sqc
Java	Cursor.sqlj
COBOL	cursor.sqb

Since REXX does not support static SQL, a sample is not provided. See "Example: Dynamic SQL Program" on page 133 for a REXX example that processes a cursor dynamically.

How the 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 For C programs that call DB2 APIs, the sqlInfoPrint function in utilapi.c is redefined as API_SQL_CHECK in utilapi.h. For C embedded SQL programs, the sqlInfoPrint function in utilemb.sqc is redefined as EMB_SQL_CHECK in utilemb.h.

Java Any SQL error is thrown as an SQLException and handled in the catch block of the application.

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 119 for the source code for this error checking utility.

C Example: CURSOR.SQC

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

EXEC SQL INCLUDE SQLCA;

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;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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
   EMB_SQL_CHECK("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
```

```
   EMB_SQL_CHECK("CLOSE CURSOR");

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

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

Java Example: Cursor.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator CursorByName(String name, short dept) ;
#sql iterator CursorByPos(String, short ) ;

class Cursor
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Cursor Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Cursor [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // Using cursors
      try
      {   CursorByName cursorByName;
        CursorByPos  cursorByPos;
```

```
        String name = null;
        short  dept=0;

        // Using the JDBC ResultSet cursor method
        System.out.println("\nUsing the JDBC ResultSet cursor method");
        System.out.println(" with a 'bind by name' cursor ...\n");

        #sql cursorByName = {
              SELECT name, dept FROM staff WHERE job='Mgr' }; 1
        while (cursorByName.next()) 2
        {   name = cursorByName.name(); 3
          dept = cursorByName.dept();

          System.out.print (" name= " + name);
          System.out.print (" dept= " + dept);
          System.out.print ("\n");
        }
        cursorByName.close(); 4


        // Using the SQLJ iterator cursor method
        System.out.println("\nUsing the SQLJ iterator cursor method");
        System.out.println(" with a 'bind by position' cursor ...\n");

        #sql cursorByPos = {
               SELECT name, dept FROM staff WHERE job='Mgr' }; 1 2
        while (true)
        {   #sql { FETCH :cursorByPos INTO :name, :dept }; 3
          if (cursorByPos.endFetch()) break;

          System.out.print (" name= " + name);
          System.out.print (" dept= " + dept);
          System.out.print ("\n");
        }
        cursorByPos.close(); 4
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        #sql { ROLLBACK };
        System.out.println("Rollback done.");
      }
    }
    catch( Exception e )
    {   System.out.println (e);
    }
  }
}
```

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.
```

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, refer to 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 5. Writing Dynamic SQL Programs" on page 127 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. Refer to 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" for select-statements which define non-updatable result tables.

There can be performance advantages for read-only cursors. For more information on read-only cursors, refer to the *Administration Guide: Implementation*.

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, refer to 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). The sample is available in the following programming languages:

C	openftch.sqc
Java	Openftch.sqlj and OpF_Curs.sqlj
COBOL	openftch.sqb

The REXX language does not support static SQL, so a sample is not provided.

How the OPENFTCH 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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

Java Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

C Example: OPENFTCH.SQC

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

EXEC SQL INCLUDE SQLCA;

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;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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
   EMB_SQL_CHECK("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;
         EMB_SQL_CHECK("UPDATE STAFF");
      }
      else
      {
         printf ("%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
         EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
         EMB_SQL_CHECK("DELETE");
      } /* endif */
   } while ( 1 );

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

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

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

Java Example: Openftch.sqlj
OpF_Curs.sqlj

```
//  PURPOSE : This file, named OpF_Curs.sqlj, contains the definition
//            of the class OpF_Curs used in the sample program Openftch.

import sqlj.runtime.ForUpdate;
#sql public iterator OpF_Curs implements ForUpdate (String, short);
```

Openftch.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Openftch
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Openftch Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception(
                "\nUsage: java Openftch [username password]\n");
      } // if - else if - else

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
```

```
  con.setAutoCommit(false);

  // Executing SQLJ positioned update/delete statements.
  try
  {   OpF_Curs forUpdateCursor;

    String name = null;
    short  dept=0;


    #sql forUpdateCursor =
    {   SELECT name, dept
            FROM staff
            WHERE job='Mgr'
    }; // #sql             1 2

    while (true)
    {   #sql
      {   FETCH :forUpdateCursor
              INTO :name, :dept
      }; // #sql                          3
      if (forUpdateCursor.endFetch()) break;

      if (dept > 40)
      {   System.out.println (
              name + " in dept. "
              + dept + " will be demoted to Clerk");
        #sql
        {   UPDATE staff SET job = 'Clerk'
            WHERE CURRENT OF :forUpdateCursor
        }; // #sql                       4
      }
      else
      {   System.out.println (
              name + " in dept. " + dept
              + " will be DELETED!");
        #sql
        {   DELETE FROM staff
            WHERE CURRENT OF :forUpdateCursor
        }; // #sql
      } // if - else
    }
    forUpdateCursor.close();   5
  }
  catch( Exception e )
  {   throw e;
  }
  finally
  {   // Rollback the transaction
    System.out.println("\nRollback the transaction...");
    #sql { ROLLBACK };
    System.out.println("Rollback done.");
  } // try - catch - finally
}
catch( Exception e )
```

```
    {   System.out.println (e);
    } // try - catch
  } // main
} // class Openftch
```

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.
```

Advanced Scrolling Techniques

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

- Scrolling Through Data that has Already Been Retrieved
- Keeping a Copy of the Data
- Retrieving the Data a Second Time
- Establishing a Position at the End of a Table
- Updating Previously Retrieved Data

Scrolling 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 and Java, however, do support a backward FETCH through read-only scrollable cursors. Refer to the *CLI Guide and Reference* and see "Creating Java Applications and Applets" on page 642 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.

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
- 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.

Establishing 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.

Updating Previously Retrieved Data

To scroll backward and update data that was retrieved previously, you can use a combination of the techniques discussed in "Scrolling Through Data that has Already Been Retrieved" on page 102 and "Updating Retrieved Data" on page 92. 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. The sample is available in the following programming languages:

C	updat.sqc
Java	Updat.sqlj
COBOL	updat.sqb
REXX	updat.cmd

How the 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.

   Java applications access `SQLCODE` and `SQLSTATE` through the methods defined for the `SQLException` object, and therefore do not need an equivalent "include SQLCA" statement.

   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.

Java and REXX applications do not need to declare host variables, except (for REXX) in the case of 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	For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.
Java	Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.
COBOL	`CHECKERR` is an external program named `checkerr.cbl`.
REXX	`CHECKERR` is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 119 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 "utilemb.h"

EXEC SQL INCLUDE SQLCA;  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;
  EMB_SQL_CHECK("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
  EMB_SQL_CHECK("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
   EMB_SQL_CHECK("UPDATE STAFF");
   printf ("All 'Mgr' have been demoted to 'Clerk'!\n" );

   strcpy (jobUpdate, "Sales");
   EXEC SQL DELETE FROM staff WHERE job = :jobUpdate;  5
   EMB_SQL_CHECK("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
   EMB_SQL_CHECK("INSERT INTO STAFF");
   printf ("New data has been inserted\n");

   EXEC SQL ROLLBACK;  7
```

```
   EMB_SQL_CHECK("ROLLBACK");
   printf( "On second thought -- changes rolled back.\n" );

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

Java Example: Updat.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;


class Updat
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }


  public static void main(String argv[])
  {   try
    {   System.out.println ("\n  Java Updat Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                  3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Updat [username password]\n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // UPDATE/DELETE/INSERT
      try
      {   String jobUpdate = null;

        jobUpdate="Clerk";
        #sql {UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr'};   4
```

```
        System.out.println("\nAll 'Mgr' have been demoted to 'Clerk'!");

        jobUpdate="Sales";
        #sql {DELETE FROM staff WHERE job = :jobUpdate};
        System.out.println("All 'Sales' people have been deleted!"); 5

        #sql {INSERT INTO staff
          VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0)}; 6
        System.out.println("New data has been inserted");
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        #sql { ROLLBACK };                                          7
        System.out.println("Rollback done.");
      }
    }
    catch (Exception e)
    {   System.out.println (e);
    }
  }
}
```

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.
```

REXX Example: UPDAT.CMD

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

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

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

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

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: updat.cmd <dbname> [<userid> <password>]"

       exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: UPDAT.CMD'

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 do
    say '--- error report ---'
    say 'ERROR occurred :' 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 do
      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, see "SQLSTATE and SQLCODE Variables in C and C++" on page 635 for C or C++ applications, "SQLSTATE and SQLCODE Variables in COBOL" on page 699 for COBOL applications, or "SQLSTATE and SQLCODE Variables in FORTRAN" on page 714 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.

Refer to the *Administrative 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.

Token Truncation in SQLCA Structure

Since tokens may be truncated in the SQLCA structure, you should not use the token info for diagnostic purposes. While you can define table and column names with lengths of up to 128 bytes, the SQLCA tokens will be truncated to 17 bytes plus a truncation terminator (>). Application logic should not depend on actual values of the `sqlerrmc` field. Refer to the *SQL Reference* for a description of the SQLCA structure, and a discussion of token truncation.

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 SQLERROR CONTINUE statement.

For a complete description of the WHENEVER statement, refer to 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:

Windows 32-bit Operating Systems
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, refer to the *Application Building Guide*.

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: UTILAPI.C" on page 120 and "COBOL Example: CHECKERR.CBL" on page 123 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: UTILAPI.C

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

EXEC SQL INCLUDE SQLCA;

/*############################################################################
**    1. SQL_CHECK section
**
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
**        1.2 - TransRollback - rolls back the transaction
############################################################################*/

/*****************************************************************************
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
*****************************************************************************/
int SqlInfoPrint( char *         appMsg,
                  struct sqlca * pSqlca,
                  int            line,
                  char *         file )
{   int   rc = 0;

    char  sqlInfo[1024];
    char  sqlInfoToken[1024];

    char  sqlstateMsg[1024];
    char  errorMsg[1024];


    if (pSqlca->sqlcode != 0 && pSqlca->sqlcode != 100)
    {   strcpy(sqlInfo, "");

        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "\n---- error report ----\n");
            strcat( sqlInfo, sqlInfoToken);
        }
        else
        {   sprintf( sqlInfoToken, "\n---- warning report ----\n");
            strcat( sqlInfo, sqlInfoToken);
        } /* endif */

        sprintf( sqlInfoToken, "  app. message      = %s\n", appMsg);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  line              = %d\n", line);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  file              = %s\n", file);
```

```
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  SQLCODE           = %ld\n", pSqlca->sqlcode);
        strcat( sqlInfo, sqlInfoToken);

        /* get error message */
        rc = sqlaintp( errorMsg, 1024, 80, pSqlca);
        /* return code is the length of the errorMsg string */
        if( rc > 0)
        {   sprintf( sqlInfoToken, "%s\n", errorMsg);
            strcat( sqlInfo, sqlInfoToken);
        }

        /* get SQLSTATE message */
        rc = sqlogstt( sqlstateMsg, 1024, 80, pSqlca->sqlstate);
        if (rc == 0)
        {   sprintf( sqlInfoToken, "%s\n", sqlstateMsg);
            strcat( sqlInfo, sqlInfoToken);
        }


        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "--- end error report ---\n");
            strcat( sqlInfo, sqlInfoToken);

            printf("%s", sqlInfo);
            return 1;
        }
        else
        {   sprintf( sqlInfoToken, "--- end warning report ---\n");
            strcat( sqlInfo, sqlInfoToken);

            printf("%s", sqlInfo);
            return 0;
        } /* endif */
    } /* endif */

    return 0;
}

/*****************************************************************************
**        1.2 - TransRollback - rolls back the transaction
*****************************************************************************/
void TransRollback( )
{   int        rc = 0;

    /* rollback the transaction */
    printf( "\nRolling back the transaction ...\n") ;
    EXEC SQL ROLLBACK;
    rc = SqlInfoPrint( "ROLLBACK", &sqlca, __LINE__, __FILE__);
    if( rc == 0)
    {   printf( "The transaction was rolled back.\n") ;
    }

}
```

Java Example: Catching SQLException

JDBC and SQLJ applications throw an SQLException when an error occurs during SQL processing. Your applications can catch and display an SQLException with the following code:

```
try {
   Statement stmt = connection.createStatement();
   int rowsDeleted = stmt.executeUpdate(
      "DELETE FROM employee WHERE empno = '000010'");
   System.out.println( rowsDeleted + " rows were deleted");
}

catch (SQLException sqle) {
   System.out.println(sqle);
}
```

For more information on handling SQLExceptions, see "SQLSTATE and SQLCODE Values in Java" on page 641.

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
      * Use application specific bound instead of BUFFER-SZ
      * 77 buffer-size     pic s9(4) comp-5 value BUFFER-SZ.
      * 77 error-buffer    pic x(BUFFER-SZ).
      * 77 state-buffer    pic x(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).

       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 occurred : ", errloc.
           display "SQLCODE : ", SQLCODE.

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

      ************************
      * GET SQLSTATE MESSAGE *
      ************************
           call "sqlggstt" using
                                by value     buffer-size
                                by value     line-width
                                by reference sqlstate
                                by reference 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.
```

REXX Example: CHECKERR Procedure

```
parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

:

call CHECKERR 'INSERT'

:

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' 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 do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0

/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
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("db2rexx")

.
.
.

call CHECKERR 'INSERT'

.
.
.

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' 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 do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

Chapter 5. Writing Dynamic SQL Programs

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 38 on page 737 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 6 on page 129 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 6. Comparing Static and Dynamic SQL

Consideration	**Likely Best Choice**
Time to run the SQL statement:	
• Less than 2 seconds	• Static
• 2 to 10 seconds	• either
• More than 10 seconds	• Dynamic
Data Uniformity	
• Uniform data distribution	• Static
• Slight non-uniformity	• either
• Highly non-uniform distribution	• Dynamic
Range (<,>,BETWEEN,LIKE) Predicates	
• Very Infrequent	• Static
• Occasional	• either
• Frequent	• Dynamic
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
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 161.
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).

Note: Java applications do not use the SQLDA structure, and therefore do not use the PREPARE or DESCRIBE statements. In JDBC applications you can use a `PreparedStatement` object and the `executeQuery()` method to generate a `ResultSet` object, which is the equivalent of a host language cursor. In SQLJ applications you can also declare an SQLJ `iterator` object with a `CursorByPos` or `CursorByName` cursor to return data from FETCH statements.

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, refer to 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 133. 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 154.

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 "Declare Cursor Statement" on page 82.

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. Table 7 shows a DECLARE statement associated with a dynamic SELECT statement.

Table 7. Declare Statement Associated with a Dynamic SELECT

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;`

Java (JDBC)	`PreparedStatement prep_string = ("SELECT tabname FROM syscat.tables`
	`WHERE tabschema = ?" );`
	`prep_string.setCursor("c1");`
	`prep_string.setString(1, host_var);`
	`ResultSet rs = prep_string.executeQuery();`

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`

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 81
- "Example: Cursor Program" on page 84
- "Using Cursors in REXX" on page 728

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. The sample is available in the following programming languages:

C	dynamic.sqc
Java	Dynamic.java
COBOL	dynamic.sqb
REXX	dynamic.cmd

How the Dynamic 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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

Java Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.

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

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

C Example: DYNAMIC.SQC

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

EXEC SQL INCLUDE SQLCA;

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;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;   3

   strcpy( parm_var, "STAFF" );
   EXEC SQL OPEN c1 USING :parm_var;   4
   EMB_SQL_CHECK("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
   EMB_SQL_CHECK("CLOSE");
```

```
   EXEC SQL COMMIT;
   EMB_SQL_CHECK("COMMIT");

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

Java Example: Dynamic.java

```
import java.sql.*;

class Dynamic
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Dynamic Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Dynamic [username password]\n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL SELECT using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT tabname FROM syscat.tables " +
          "WHERE tabname <> ? " +
          "ORDER BY 1"); 2
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");                                    3
      pstmt1.setString(1, "STAFF");
      ResultSet rs = pstmt1.executeQuery();                          4

      System.out.print("\n");
      while( rs.next() )                                             5
```

```
      {   String tableName = rs.getString("tabname");
        System.out.println("Table = " + tableName);
      };

      rs.close();
      pstmt1.close();                                                    7
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        con.rollback();
        System.out.println("Rollback done.");
      }
    }
    catch( Exception e )
    {   System.out.println(e);
    }
  }
}
```

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
      -       " ORDER BY 1
      -       " 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.
```

REXX Example: DYNAMIC.CMD

```
/* REXX DYNAMIC.CMD */

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

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

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: dynamic.cmd <dbname> [<userid> <password>]"

       exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: DYNAMIC'

st = "SELECT tabname FROM syscat.tables WHERE tabname <> ? ORDER BY 1"
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 do
    say '--- error report ---'
    say 'ERROR occurred :' 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 do
      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 2. There are two types of SQLVAR entries: base SQLVARs, and secondary SQLVARs. For information about the two types, refer to the *SQL Reference*.

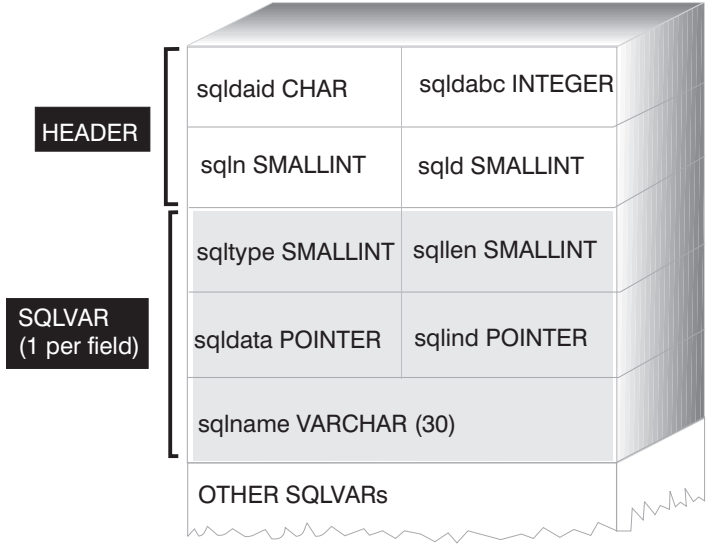


Figure 2. The SQL Descriptor Area (SQLDA)

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, refer to 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".

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 on 32-bit platforms, and 56 bytes in length on 64-bit platforms.

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 * sizeof(struct sqlvar))
```

Note: On 64-bit platforms, `sizeof(struct sqlvar)` and `sizeof(struct sqlvar2)` returns 56. On 32-bit platforms, `sizeof(struct sqlvar)` and `sizeof(struct sqlvar2)` returns 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 10. Using the Object-Relational Capabilities" on page 275.

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 154.)

In the C/C++ programming language, a macro is provided to facilitate SQLDA allocation. With the exception of the HP-UX platform, this macro has the following format:

```
#define SQLDASIZE(n) (offsetof(struct sqlda, sqlvar) + (n) × sizeof(struct sqlvar))
```

On the HP-UX platform, the 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` `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


            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_sqlname1
      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_sqlname1                                    )
      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 154.) See Table 8 for a convenient cross-reference listing showing how the numeric values and symbolic names are related.

Table 8. 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

Table 8. DB2 V2 SQLDA SQL Types (continued). Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name[1]
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

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 154.

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 the *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 143
- "Preparing the Statement Using the Minimum SQLDA Structure" on page 144
- "Allocating an SQLDA with Sufficient SQLVAR Entries" on page 145
- "Describing the SELECT Statement" on page 146
- "Acquiring Storage to Hold a Row" on page 146
- "Processing the Cursor" on page 147.

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: The example adhoc.sqc exists for C only.

How the 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 `EMB_SQL_CHECK` macro/function is an error checking utility which is external to this program. For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h. See "Using GET ERROR MESSAGE in Example Programs" on page 119 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 `utilemb.sqc`. These include:

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 "utilemb.h"

#ifdef DB268K
  /* Need to include ASLM for 68K applications */
  #include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA ;  2

#define SQLSTATE sqlca.sqlstate

int process_statement( char * ) ;

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

   int rc ;
   char sqlInput[256] ;
   char st[1024] ;

   EXEC SQL BEGIN DECLARE SECTION ;  3
      char userid[9] ;
      char passwd[19] ;
   EXEC SQL END DECLARE SECTION ;

#ifdef DB268K
  /*
   Before making any API calls for 68K environment,
   need to initial the Library Manager
   */
   InitLibraryManager(0,kCurrentZone,kNormalMemory) ;
   atexit(CleanupLibraryManager) ;
#endif

   printf( "Sample C program : ADHOC interactive SQL\n" ) ;

   /* Initialize the connection to a database. */
   if ( argc == 1 ) {
      EXEC SQL CONNECT TO sample ;
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;
   }
   else if ( argc == 3 ) {
      strcpy( userid, argv[1] ) ;
      strcpy( passwd, argv[2] ) ;
      EXEC SQL CONNECT TO sample USER :userid USING :passwd ;  4
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;  5
   }
   else {
      printf( "\nUSAGE: adhoc [userid passwd]\n\n" ) ;
```

```
      return( 1 ) ;
   } /* endif */

   printf( "Connected to database SAMPLE\n" ) ;

   /* Enter the continuous command line loop. */
   *sqlInput = '\0' ;
   while ( ( *sqlInput != 'q' ) && ( *sqlInput != 'Q' ) ) { 6

      printf( "Enter an SQL statement or 'quit' to Quit :\n" ) ;
      gets( sqlInput ) ;

      if ( ( *sqlInput == 'q' ) || ( *sqlInput == 'Q' ) ) break ;

      if ( *sqlInput == '\0' ) { /* Don't process the statement */
         printf( "No characters entered.\n" ) ;
         continue ;
      }

      strcpy( st, sqlInput ) ;
      while ( sqlInput[strlen( sqlInput ) - 1] == '\\' ) {
         st[strlen( st ) - 1] = '\0' ;
         gets( sqlInput ) ;
         strcat( st, sqlInput ) ;
      }

      /* Process the statement. */
      rc = process_statement( st ) ;

   }

   printf( "Enter 'c' to COMMIT or Any Other key to ROLLBACK the transaction :\n" ) ;
   gets( sqlInput ) ;
   if ( ( *sqlInput == 'c' ) || ( *sqlInput == 'C' ) ) {
      printf( "COMMITING the transactions.\n" ) ;
      EXEC SQL COMMIT ;  7
      EMB_SQL_CHECK( "COMMIT" ) ;
   }
   else { /* assume that the transaction is to be rolled back */
      printf( "ROLLING BACK the transactions.\n" ) ;
      EXEC SQL ROLLBACK ; 8
      EMB_SQL_CHECK( "ROLLBACK" ) ;
   }

   EXEC SQL CONNECT RESET ; 9
   EMB_SQL_CHECK( "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 ) {

   int counter = 0 ;
   struct sqlda * sqldaPointer ;
   short sqlda_d ;

   EXEC SQL BEGIN DECLARE SECTION ; 3
      char st[1024] ;
   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 ;
   /* EMB_SQL_CHECK( "PREPARE" ) ; */

   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 occurred. Check the SQLCA. */
      EMB_SQL_CHECK( "DESCRIBE" ) ;
   } /* end if */

   sqlda_d = sqldaPointer->sqld ;
   free( sqldaPointer ) ;

   if ( sqlda_d > 0 ) { 12

      /* this is a SELECT statement, a number of columns
         are present in the SQLDA */

      if ( SQLCODE == SQL_RC_W236 || SQLCODE == 0)
         /* this out only needs a SINGLE SQLDA */
         init_da( &sqldaPointer, sqlda_d ) ;

      if ( SQLCODE == SQL_RC_W237 ||
           SQLCODE == SQL_RC_W238 ||
           SQLCODE == SQL_RC_W239 )
         /* this output contains columns that need a DOUBLED SQLDA */
         init_da( &sqldaPointer, sqlda_d * 2 ) ;

      /* need to reassign the SQLDA with the correct number
         of columns to the SQL statement */
```

```
      EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;
      EMB_SQL_CHECK( "DESCRIBE" ) ;

      /* 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
      EMB_SQL_CHECK( "OPEN" ) ;

      EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer; 16
      EMB_SQL_CHECK( "FETCH" ) ;

      /* 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
      EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
      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 ;
      EMB_SQL_CHECK( "Executing the SQL statement" ) ;
   }  /* end if */

   return( 0 ) ;

}     /* 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. The sample is available in the following programming languages:

C varinp.sqc

Java Varinp.java

COBOL varinp.sqb

How the VARINP 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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C

embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

Java Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

C Example: VARINP.SQC

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

EXEC SQL INCLUDE SQLCA;

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

   strcpy (st, "SELECT name, dept FROM staff ");
   strcat (st, "WHERE job = ? FOR UPDATE OF job");
   EXEC SQL PREPARE s1 FROM :st; 1
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1; 2

   strcpy (parm_var, "Mgr");
   EXEC SQL OPEN c1 USING :parm_var; 3
   EMB_SQL_CHECK("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
      EMB_SQL_CHECK("EXECUTE");
   } while ( 1 );

   EXEC SQL CLOSE c1;  7
   EMB_SQL_CHECK("CLOSE CURSOR");

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

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

Java Example: Varinp.java

```
import java.sql.*;

class Varinp
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("\n  Error loading DB2 Driver...\n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Varinp Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("\nUsage: java Varinp [username password]\n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job");  1
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");                                           2
      pstmt1.setString(1, "Mgr");
      ResultSet rs = pstmt1.executeQuery();                                 3

      PreparedStatement pstmt2 = con.prepareStatement(
          "UPDATE staff SET job = ? WHERE CURRENT OF c1");                  4
      pstmt2.setString(1, "Clerk");
```

```
      System.out.print("\n");
      while( rs.next() )                                                    5
      {   String name = rs.getString("name");
        short  dept = rs.getShort("dept");
        System.out.println(name + " in dept. " + dept
          + " will be demoted to Clerk");

        pstmt2.executeUpdate();                                             6
      };

      rs.close();
      pstmt1.close();                                                       7
      pstmt2.close();
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        con.rollback();
        System.out.println("Rollback done.");
      }
    }
    catch( Exception e )
    {   System.out.println(e);
    }
  }
}
```

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.
```

The DB2 Call Level Interface (CLI) Differences Between DB2 CLI and Embedded SQL

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). Refer to for a cross reference of all DB2 CLI SQLSTATEs.

- DB2 CLI supports scrollable cursors. With scrollable cursors, you can scroll through a static 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 38 on page 737 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 a server 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. It is much easier to use static SQL in embedded applications.

For more information on using static SQL in CLI applications, refer to the Web page at:

```
http://www.ibm.com/software/data/db2/udb/staticcli
```

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*.

For more information on using static SQL in CLI applications, refer to the Web page at:

`http://www.ibm.com/software/data/db2/udb/staticcli`

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 6. Common DB2 Application Techniques

DB2 enables you to use embedded SQL to handle common database application development problems.

Generated columns
Rather than using cumbersome insert and update triggers, DB2 enables you to include generated columns in your tables using the GENERATED ALWAYS AS clause. Generated columns provide automatically updated values derived from an SQL expression.

Identity columns
DB2 application developers often need to create a primary key for every row in a table. If you create a table that uses an identity column for the primary key, DB2 automatically inserts a unique value. When you use identity columns, your applications can benefit from increased performance due to a reduction in lock contention.

Sequence objects
Sequence objects are database objects that generate sequential values for use in any SQL statement.

Declared temporary tables
Declared temporary tables are similar to regular tables, but persist only as long as the database connection and are not subject to locking or logging. If your application creates tables to process large amounts of data and drops those tables once the application has finished manipulating that data, consider using declared temporary tables. Declared temporary tables can increase the performance of your application and, for applications designed for concurrent users, simplify your application development effort.

External savepoints
While COMMIT and ROLLBACK statements enable you to control the behavior of an entire transaction, savepoints enable you to exercise more granular control within transactions. Savepoint blocks group

several SQL statements together. If one of the sub-statements in the savepoint block results in an error, you can roll back just the failing sub-statement and complete the work of the other sub-statements.

Generated Columns

A generated column is a column that derives the values for each row from an expression, rather than from an insert or update operation. While combining an update trigger and an insert trigger can achieve a similar effect, using a generated column guarantees that the derived value is consistent with the expression.

To create a generated column in a table, use the GENERATED ALWAYS AS clause for the column and include the expression from which the value for the column will be derived. You can include the GENERATED ALWAYS AS clause in ALTER TABLE or CREATE TABLE statements. The following example creates a table with two regular columns, "c1" and "c2", and two generated columns, "c3" and "c4", that are derived from the regular columns of the table.

```
CREATE TABLE T1(c1 INT, c2 DOUBLE,
                c3 DOUBLE GENERATED ALWAYS AS (c1 + c2),
                c4 GENERATED ALWAYS AS
                  (CASE
                     WHEN c1 > c2 THEN 1
                     ELSE NULL
                   END)
               );
```

For more information on using generated columns to improve the performance of your applications, refer to the *Administration Guide*. For more information on creating generated columns, refer to the CREATE TABLE statement syntax in the *SQL Reference*.

Identity Columns

Identity columns provide DB2 application developers with an easy way of automatically generating a numeric column value for every row in a table. You can have this value generated as a unique value, then define the identity column as the primary key for the table. To create an identity column, include the IDENTITY clause in the CREATE TABLE or ALTER TABLE statement.

Use identity columns in your applications to avoid the concurrency and performance problems that can occur when an application generates its own unique counter outside the database. When you do not use identity columns to automatically generate unique primary keys, a common design is to store a counter in a table with a single row. Each transaction then locks this table,

increments the number, and then commits the transaction to unlock the counter. Unfortunately, this design only allows a single transaction to increment the counter at a time.

In contrast, if you use an identity column to automatically generate primary keys, the application can achieve much higher levels of concurrency. With identity columns, DB2 maintains the counter so that transactions do not have to lock the counter. Applications that use identity columns can perform better because an uncommitted transaction that has incremented the counter does not prevent other subsequent transactions from also incrementing the counter.

The counter for the identity column is incremented or decremented independently of the transaction. If a given transaction increments an identity counter two times, that transaction may see a gap in the two numbers that are generated because there may be other transactions concurrently incrementing the same identity counter.

An identity column may appear to have generated gaps in the counter, as the result of a transaction that was rolled back, or because the database cached a range of values that have been deactivated (normally or abnormally) before all the cached values were assigned.

To retrieve the generated value after inserting a new row into a table with an identity column, use the `identity_val_local()` function.

For more information on identity columns, refer to the *Administration Guide*. For more information on the IDENTITY clause of the CREATE TABLE and ALTER TABLE statements, refer to the *SQL Reference*.

Generating Sequential Values

Generating sequential values is a common database application development problem. The best solution to that problem is to use sequence objects and sequence expressions in SQL. Each *sequence object* is a uniquely named database object that can be accessed only by sequence expressions. There are two *sequence expressions*: the PREVVAL expression and the NEXTVAL expression. The PREVVAL expression returns the value most recently generated in the application process for the specified sequence object. Any NEXTVAL expressions occuring in the same statement as the PREVAL expression have no effect on the value generated by the PREVAL expression in that statement. The NEXTVAL sequence expression increments the value of the sequence object and returns the new value of the sequence object.

To create a sequence object, issue the CREATE SEQUENCE statement. For example, to create a sequence object called id_values using the default attributes, issue the following statement:

```
CREATE SEQUENCE id_values
```

To generate the first value in the application session for the sequence object, issue a VALUES statement using the NEXTVAL expression:

```
VALUES NEXTVAL FOR id_values

1
-----------
          1

            1 record(s) selected.
```

To display the current value of the sequence object, issue a VALUES statement using the PREVVAL expression:

```
VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.
```

You can repeatedly retrieve the current value of the sequence object, and the value that the sequence object returns does not change until you issue a NEXTVAL expression. In the following example, the PREVVAL expression returns a value of 1, until the NEXTVAL expression in the application process increments the value of the sequence object:

```
VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.

VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.

VALUES NEXTVAL FOR id_values

1
-----------
          2

            1 record(s) selected.

VALUES PREVVAL FOR id_values
```

```
1
-----------
          2

            1 record(s) selected.
```

To update the value of a column with the next value of the sequence object, include the NEXTVAL expression in the UPDATE statement, as follows:

```
UPDATE staff
  SET id = NEXTVAL FOR id_values
  WHERE id = 350
```

To insert a new row into a table using the next value of the sequence object, include the NEXTVAL expression in the INSERT statement, as follows:

```
INSERT INTO staff (id, name, dept, job)
  VALUES (NEXTVAL FOR id_values, 'Kandil', 51, 'Mgr')
```

For more information on the PREVVAL and NEXTVAL expressions, refer to the *SQL Reference*.

Controlling Sequence Behavior

You can tailor the behavior of sequence objects to meet the needs of your application. You change the attributes of a sequence object when you issue the CREATE SEQUENCE statement to create a new sequence object, and when you issue the ALTER SEQUENCE statement for an existing sequence object. Following are some of the attributes of a sequence object that you can specify:

Data type
: The AS clause of the CREATE SEQUENCE statement specifies the numeric data type of the sequence object. The data type, as specified in the "SQL Limits" appendix of the *SQL Reference*, determines the possible minimum and maximum values of the sequence object. You cannot change the data type of a sequence object; instead, you must drop the sequence object by issuing the DROP SEQUENCE statement and issuing a CREATE SEQUENCE statement with the new data type.

Start value
: The START WITH clause of the CREATE SEQUENCE statement sets the initial value of the sequence object. The RESTART WITH clause of the ALTER SEQUENCE statement resets the value of the sequence object to a specified value.

Minimum value
: The MINVALUE clause sets the minimum value of the sequence object.

Maximum value
The MAXVALUE clause sets the maximum value of the sequence object.

Increment value
The INCREMENT BY clause sets the value that each NEXTVAL expression adds to the current value of the sequence object. To decrement the value of the sequence object, specify a negative value.

Sequence cycling
The CYCLE clause causes the value of a sequence object that reaches its maximum or minimum value to generate its respective minimum value or maximum value on the following NEXTVAL expression.

For example, to create a sequence object called id_values that starts with a minimum value of 0, has a maximum value of 1000, increments by 2 with each NEXTVAL expression, and returns to its minimum value when the maximum value is reached, issue the following statement:

```
CREATE SEQUENCE id_values
  START WITH 0
  INCREMENT BY 2
  MAXVALUE 1000
  CYCLE
```

For more information on the CREATE SEQUENCE and ALTER SEQUENCE statements, refer to the *SQL Reference*.

Improving Performance with Sequence Objects

Like identity columns, using sequence objects to generate values generally improves the performance of your applications in comparison to alternative approaches. The alternative to sequence objects is to create a single-column table that stores the current value and incrementing that value with either a trigger or under the control of the application. In a distributed environment where applications concurrently access the single-column table, the locking required to force serialized access to the table can seriously affect performance.

Sequence objects avoid the locking issues that are associated with the single-column table approach and can cache sequence values in memory to improve DB2 response time. To maximize the performance of applications that use sequence objects, ensure that your sequence object caches an appropriate amount of sequence values. The CACHE clause of the CREATE SEQUENCE and ALTER SEQUENCE statements specifies the maximum number of sequence values that DB2 generates and stores in memory.

If your sequence object must generate values in order, without introducing gaps in that order due to a system failure or database deactivation, use the ORDER and NO CACHE clauses in the CREATE SEQUENCE statement. The

NO CACHE clause guarantees that no gaps appear in the generated values at the cost of some of your application's performance because it forces your sequence object to write to the database log every time it generates a new value. Note that gaps can still appear due to transactions that rollback and do not actually use that sequence value that they requested.

Comparing Sequence Objects and Identity Columns

Although sequence objects and identity columns appear to serve similar purposes for DB2 applications, there is an important difference. An identity column automatically generates values for a column in a single table. A sequence object generates sequential values upon request that can be used in any SQL statement.

Declared Temporary Tables

A *declared temporary table* is a temporary table that is only accessible to SQL statements that are issued by the application which created the temporary table. A declared temporary table does not persist beyond the duration of the connection of the application to the database.

Use declared temporary tables to potentially improve the performance of your applications. When you create a declared temporary table, DB2 does not insert an entry into the system catalog tables, and therefore your server does not suffer from catalog contention issues. In comparison to regular tables, DB2 does not lock declared temporary tables or their rows, and does not log declared temporary tables or their contents. If your current application creates tables to process large amounts of data and drops those tables once the application has finished manipulating that data, consider using declared temporary tables instead of regular tables.

If you develop applications written for concurrent users, your applications can take advantage of declared temporary tables. Unlike regular tables, declared temporary tables are not subject to name collision. For each instance of the application, DB2 can create a declared temporary table with an identical name. For example, to write an application for concurrent users that uses regular tables to process large amounts of temporary data, you must ensure that each instance of the application uses a unique name for the regular table that holds the temporary data. Typically, you would create another table that tracks the names of the tables that are in use at any given time. With declared temporary tables, simply specify one declared temporary table name for your temporary data. DB2 guarantees that each instance of the application uses a unique table.

To use a declared temporary table, perform the following steps:

Step 1. Ensure that a USER TEMPORARY TABLESPACE exists. If a USER TEMPORARY TABLESPACE does not exist, issue a CREATE USER TEMPORARY TABLESPACE statement.

Step 2. Issue a DECLARE GLOBAL TEMPORARY TABLE statement in your application.

The schema for declared temporary tables is always SESSION. To use the declared temporary table in your SQL statements, you must refer to the table using the SESSION schema qualifier either explicitly or by using a DEFAULT schema of SESSION to qualify any unqualified references. In the following example, the table name is always qualified by the schema name SESSION when you create a declared temporary table named TT1 with the following statement:

```
DECLARE GLOBAL TEMPORARY TABLE TT1
```

To select the contents of the *column1* column from the declared temporary table created in the previous example, use the following statement:

```
SELECT column1 FROM SESSION.TT1;
```

Note that DB2 also enables you to create persistent tables with the SESSION schema. If you create a persistent table with the qualified name SESSION.TT3, you can then create a declared temporary table with the qualified name SESSION.TT3. In this situation, DB2 always resolves references to persistent and declared temporary tables with identical qualified names to the declared temporary table. To avoid confusing persistent tables with declared temporary tables, you should not create persistent tables using the SESSION schema.

If you create an application that includes a static SQL reference to a table, view, or alias qualified with the SESSION schema, the DB2 precompiler does not compile that statement at bind time and marks the statement as "needing compilation". At run time, DB2 compiles the statement. This behavior is called *incremental binding*. DB2 automatically performs incremental binding for static SQL references to tables, views, and aliases qualified with the SESSION schema. You do not need to specify the VALIDATE RUN option on the BIND or PRECOMPILE command to enable incremental binding for these statements.

If you issue a ROLLBACK statement for a transaction that includes a DECLARE GLOBAL TEMPORARY TABLE statement, DB2 drops the declared temporary table. If you issue a DROP TABLE statement for a declared temporary table, issuing a ROLLBACK statement for that transaction only restores an empty declared temporary table. A ROLLBACK of a DROP TABLE statement does not restore the rows that existed in the declared temporary table.

The default behavior of a declared temporary table is to delete all rows from the table when you commit a transaction. However, if one or more WITH HOLD cursors are still open on the declared temporary table, DB2 does not delete the rows from the table when you commit a transaction. To avoid deleting all rows when you commit a transaction, create the temporary table using the ON COMMIT PRESERVE ROWS clause in the DECLARE GLOBAL TEMPORARY TABLE statement.

If you modify the contents of a declared temporary table using an INSERT, UPDATE, or DELETE statement within a transaction, and roll back that transaction, DB2 deletes all of the rows of the declared temporary table. If you attempt to modify the contents of a declared temporary table using an INSERT, UPDATE, or DELETE statement, and the statement fails, DB2 deletes all of the rows of the declared temporary table.

In a partitioned environment, when a node failure is encountered, all declared temporary tables that have a partition on the failed node become unusable. Any subsequent access to those unusable declared temporary tables returns an error (SQL1477N). When your application encounters an unusable declared temporary table the application can either drop the table or recreate the table by specifying the WITH REPLACE clause in the DECLARE GLOBAL TEMPORARY TABLE statement.

Declared temporary tables are subject to a number of restrictions. For example, you cannot define indexes, aliases, or views for declared temporary tables. You cannot use IMPORT and LOAD to populate declared temporary tables. For the complete syntax of the DECLARE GLOBAL TEMPORARY TABLE statement, and a complete list of the restrictions on declared temporary tables, refer to the *SQL Reference*.

Controlling Transactions with Savepoints

Application savepoints provide control over the work performed by a subset of SQL statements in a transaction or unit of work. Within your application you can set a savepoint, and later either release the savepoint or roll back the work performed since you set the savepoint. You can use multiple savepoints within a single transaction, however, you cannot nest savepoints. The following example demonstrates the use of two savepoints within a single transaction to control the behavior of an application:

Example of an order using application savepoints:

```
INSERT INTO order ...
INSERT INTO order_item ... lamp

-- set the first savepoint in the transaction
SAVEPOINT before_radio ON ROLLBACK RETAIN CURSORS
  INSERT INTO order_item ... Radio
```

```
  INSERT INTO order_item ... Power Cord
  -- Pseudo-SQL:
  IF SQLSTATE = "No Power Cord"
    ROLLBACK TO SAVEPOINT before_radio
RELEASE SAVEPOINT before_radio

-- set the second savepoint in the transaction
SAVEPOINT before_checkout ON ROLLBACK RETAIN CURSORS
  INSERT INTO order ... Approval
  -- Pseudo-SQL:
  IF SQLSTATE = "No approval"
    ROLLBACK TO SAVEPOINT before_checkout

-- commit the transaction, which releases the savepoint
COMMIT
```

In the preceding example, the first savepoint enforces a dependency between two data objects where the dependency is not intrinsic to the objects themselves. You would not use referential integrity to describe the above relationship between radios and power cords since one can exist without the other. However, you do not want to ship the radio to the customer without a power cord. You also would not want to cancel the order of the lamp by rolling back the entire transaction because there are no power cords for the radio. Application savepoints provide the granular control you need to complete this order.

When you issue a ROLLBACK TO SAVEPOINT statement, the corresponding savepoint is not automatically released. Any subsequent SQL statements are associated with that savepoint, until the savepoint is released either explicitly with a RELEASE SAVEPOINT statement or implicitly by ending the transaction or unit of work. This means that you can issue multiple ROLLBACK TO SAVEPOINT statements for a single savepoint.

Savepoints give you better performance and a cleaner application design than using multiple COMMIT and ROLLBACK statements. When you issue a COMMIT statement, DB2 must do some extra work to commit the current transaction and start a new transaction. Savepoints allow you to break a transaction into smaller units or steps without the added overhead of multiple COMMIT statements. The following example demonstrates the performance penalty incurred by using multiple transactions instead of savepoints:

Example of an order using multiple transactions::

```
INSERT INTO order ...
INSERT INTO order_item ... lamp
-- commit current transaction, start new transaction
COMMIT

INSERT INTO order_item ... Radio
INSERT INTO order_item ... Power Cord
```

```
-- Pseudo-SQL:
IF SQLSTATE = "No Power Cord"
  -- roll back current transaction, start new transaction
  ROLLBACK
ELSE
  -- commit current transaction, start new transaction
  COMMIT

INSERT INTO order ... Approval
-- Pseudo-SQL:
IF SQLSTATE = "No approval"
  -- roll back current transaction, start new transaction
  ROLLBACK
ELSE
  -- commit current transaction, start new transaction
  COMMIT
```

Another drawback of multiple commit points is that an object might be committed and therefore visible to other applications before it is fully completed. In on page 184 the order is available to another user before all the items have been added, and worse, before it has been approved. Using application savepoints avoids this exposure to 'dirty data' while providing granular control over an operation.

Comparing application savepoints to compound SQL blocks

Savepoints offer the following advantages over compound SQL blocks:

- enhanced control of transactions
- less locking contention
- improved integration with application logic

Compound SQL blocks can either be ATOMIC or NOT ATOMIC. If a statement within an ATOMIC compound SQL block fails, the entire compound SQL block is rolled back. If a statement within a NOT ATOMIC compound SQL block fails, the commit or roll back of the transaction, including the entire compound SQL block, is controlled by the application. In comparison, if a statement within the scope of a savepoint fails, the application can roll back all of the statements in the scope of the savepoint, but commit the work performed by statements outside of the scope of the savepoint. This option is illustrated in on page 183. If the work of the savepoint is rolled back, the work of the two INSERT statements before the savepoint is committed. Alternately, the application can commit the work performed by all of the statements in the transaction, including the statements within the scope of the savepoint.

When you issue a compound SQL block, DB2 simultaneously acquires the locks needed for the entire compound SQL block of statements. When you set an application savepoint, DB2 acquires locks as each statement in the scope of the savepoint is issued. The locking behavior of savepoints can lead to

significantly less locking contention than compound SQL blocks, so unless your application requires the locking performed by compound SQL statements, it may be best to use savepoints.

Compound SQL blocks execute a complete set of statements as a single statement. An application cannot use control structures or functions to add statements to a compound SQL block. In comparison, when you set an application savepoint, your application can issue SQL statements within the scope of the savepoint by calling other application functions or methods, through control structures such as while loops, or with dynamic SQL statements. Application savepoints give you the freedom to integrate your SQL statements with your application logic in an intuitive way.

For example, in 186, the application sets a savepoint and issues two INSERT statements within the scope of the savepoint. The application uses an IF statement that, when true, calls the function add_batteries(). The add_batteries() function issues an SQL statement that in this context is included within the scope of the savepoint. Finally, the application either rolls back the work performed within the savepoint (including the SQL statement issued by the add_batteries() function), or commits the work performed in the entire transaction:

Example of integrating savepoints and SQL statements within application logic:

```
void add_batteries()
{
  -- the work performed by the following statement
  -- is controlled by the savepoint set in main()
  INSERT INTO order_item ... Batteries
}

void main(int argc, char[] *argv)
{
  INSERT INTO order ...
  INSERT INTO order_item ... lamp

  -- set the first savepoint in the transaction
  SAVEPOINT before_radio ON ROLLBACK RETAIN CURSORS
    INSERT INTO order_item ... Radio
    INSERT INTO order_item ... Power Cord

    if (strcmp(Radio..power_source(), "AC/DC"))
    {
      add_batteries();
    }

    -- Pseudo-SQL:
```

```
    IF SQLSTATE = "No Power Cord"
      ROLLBACK TO SAVEPOINT before_radio
  COMMIT
}
```

List of Savepoint SQL Statements

The following SQL statements enable you to create and control savepoints:

SAVEPOINT
To set a savepoint, issue a SAVEPOINT SQL statement. To improve the clarity of your code, you can choose a meaningful name for the savepoint. For example:

```
SAVEPOINT savepoint1 ON ROLLBACK RETAIN CURSORS
```

RELEASE SAVEPOINT
To release a savepoint, issue a RELEASE SAVEPOINT SQL statement. If you do not explicitly release a savepoint with a RELEASE SAVEPOINT SQL statement, it is released at the end of the transaction. For example:

```
RELEASE SAVEPOINT savepoint1
```

ROLLBACK TO SAVEPOINT
To rollback to a savepoint, issue a ROLLBACK TO SAVEPOINT SQL statement. For example:

```
ROLLBACK TO SAVEPOINT
```

For the complete syntax of the SAVEPOINT, RELEASE SAVEPOINT, and ROLLBACK TO SAVEPOINT statements, refer to the *SQL Reference*.

Savepoint Restrictions

DB2 Universal Database places the following restrictions on your use of savepoints in applications:

Atomic compound SQL
DB2 does not enable you to use savepoints within atomic compound SQL. You cannot use atomic compound SQL within a savepoint.

Nested Savepoints
DB2 does not support the use of a savepoint within another savepoint.

Triggers
DB2 does not support the use of savepoints in triggers.

SET INTEGRITY statement
Within a savepoint, DB2 treats SET INTEGRITY statements as DDL statements. For more information on using DDL in savepoints, see "Savepoints and Data Definition Language (DDL)" on page 188.

Savepoints and Data Definition Language (DDL)

DB2 enables you to include DDL statements within a savepoint. If the application successfully releases a savepoint that executes DDL statements, the application can continue to use the SQL objects created by the DDL. However, if the application issues a ROLLBACK TO SAVEPOINT statement for a savepoint that executes DDL statements, DB2 marks any cursors that depend on the effects of those DDL statements as invalid.

In the following example, the application attempts to fetch from three previously opened cursors after issuing a ROLLBACK TO SAVEPOINT statement:

```
SAVEPOINT savepoint_name;
   PREPARE s1 FROM 'SELECT FROM t1';
   --issue DDL statement for t1
      ALTER TABLE t1 ADD COLUMN...
   PREPARE s2 FROM 'SELECT FROM t2';
   --issue DDL statement for t3
      ALTER TABLE t3 ADD COLUMN...
   PREPARE s3 FROM 'SELECT FROM t3';
   OPEN c1 USING s1;
   OPEN c2 USING s2;
   OPEN c3 USING s3;
ROLLBACK TO SAVEPOINT
FETCH c1; --invalid (SQLCODE -910)
FETCH c2; --successful
FETCH c3; --invalid (SQLCODE -910)
```

At the ROLLBACK TO SAVEPOINT statement, DB2 marks cursors “c1” and “c3” as invalid because the SQL objects on which they depend have been manipulated by DDL statements within the savepoint. However, a FETCH using cursor “c2” from the example is successful after the ROLLBACK TO SAVEPOINT statement.

You can issue a CLOSE statement to close invalid cursors. If you issue a FETCH against an invalid cursor, DB2 returns SQLCODE –910. If you issue an OPEN statement against an invalid cursor, DB2 returns SQLCODE –502. If you issue an UPDATE or DELETE WHERE CURRENT OF statement against an invalid cursor, DB2 returns SQLCODE –910.

Within savepoints, DB2 treats tables with the NOT LOGGED INITIALLY property and temporary tables as follows:

NOT LOGGED INITIALLY tables

Within a savepoint, you can create a table with the NOT LOGGED INITIALLY property, or alter a table to have the NOT LOGGED INITIALLY property. For these savepoints, however, DB2 treats ROLLBACK TO SAVEPOINT statements as ROLLBACK WORK statements and rolls back the entire transaction.

DECLARE TEMPORARY TABLE inside savepoint
If a temporary table is declared within a savepoint, a ROLLBACK TO SAVEPOINT statement drops the temporary table.

DECLARE TEMPORARY TABLE outside savepoint
If a temporary table is declared outside a savepoint, a ROLLBACK TO SAVEPOINT statement does not drop the temporary table.

Savepoints and Buffered Inserts

To improve the performance of DB2 applications, you can use buffered inserts in your applications by precompiling or binding with the INSERT BUF option. If your application takes advantage of both buffered inserts and savepoints, DB2 flushes the buffer before executing SAVEPOINT, RELEASE SAVEPOINT, OR ROLLBACK TO SAVEPOINT statements.

For more information on using buffered inserts in an application, see "Using Buffered Inserts" on page 557. For more information on precompiling and binding applications, refer to the *Command Reference*.

Using Savepoints with Cursor Blocking

If your application uses savepoints, consider preventing cursor clocking by precompiling or binding the application with the precompile option BLOCKING NO. While blocking cursors can improve the performance of your application by pre-fetching multiple rows, the data returned by an application that uses savepoints and blocking cursors may not reflect data that has been committed to the database.

If you do not precompile the application using BLOCKING NO, and your application issues a FETCH statement after a ROLLBACK TO SAVEPOINT has occurred, the FETCH statement may retrieve deleted data. For example, assume that the application containing the following SQL is precompiled without the BLOCKING NO option:

```
CREATE TABLE t1(c1 INTEGER);
DECLARE CURSOR c1 AS 'SELECT c1 FROM t1 ORDER BY c1';
INSERT INTO t1 VALUES (1);
SAVEPOINT showFetchDelete;
   INSERT INTO t1 VALUES (2);
   INSERT INTO t1 VALUES (3);
   OPEN CURSOR c1;
   FETCH c1; --get first value and cursor block
   ALTER TABLE t1... --add constraint
ROLLBACK TO SAVEPOINT;
FETCH c1; --retrieves second value from cursor block
```

When your application issues the first FETCH on table "t1", the DB2 server sends a block of column values (1, 2 and 3) to the client application. These column values are stored locally by the client. When your application issues the ROLLBACK TO SAVEPOINT SQL statement, column values '2' and '3' are

deleted from the table. After the ROLLBACK TO SAVEPOINT statement, the next FETCH from the table returns column value '2' even though that value no longer exists in the table. The application receives this value because it takes advantage of the cursor blocking option to improve performance and accesses the data that it has stored locally.

For more information on precompiling and binding applications, refer to the *Command Reference*.

Savepoints and XA Compliant Transaction Managers

If there are any active savepoints in an application when an XA compliant transaction manager issues an XA_END request, DB2 issues a RELEASE SAVEPOINT statement.

Part 3. Stored Procedures

Chapter 7. Stored Procedures

Stored Procedure Overview

Use stored procedures to improve the performance of your client/server applications. A *stored procedure* is a function in a shared library accessible to the database server. Stored procedures access the database locally and return information to *client applications*. A stored procedure saves the overhead of having a remote application pass multiple SQL statements to the server. With a single CALL statement, a client application invokes the stored procedure, which then performs the database access work and returns the results to the client application.

You can write stored procedures using SQL, called *SQL procedures*. For more information on writing SQL procedures, see "Chapter 8. Writing SQL Procedures" on page 247. You can also write stored procedures using languages such as C or Java. You do not have to write client applications in the same language as the stored procedure. When the language of the client application and the stored procedure differ, DB2 transparently passes the values between the client and the stored procedure.

You can use the DB2 Stored Procedure Builder (SPB) to help develop Java or SQL stored procedures. You can integrate SPB with popular application development tools, including Microsoft Visual Studio and IBM Visual Age for Java, or you can use it as a standalone utility. To help you create your stored procedures, SPB provides design assistants that guide you through basic design patterns, help you create SQL queries, and estimate the performance cost of invoking a stored procedure.

For more information on the DB2 Stored Procedure Builder, see "Chapter 9. IBM DB2 Stored Procedure Builder" on page 269.

Advantages of Stored Procedures

Figure 3 shows how a normal database manager application accesses a database located on a database server.

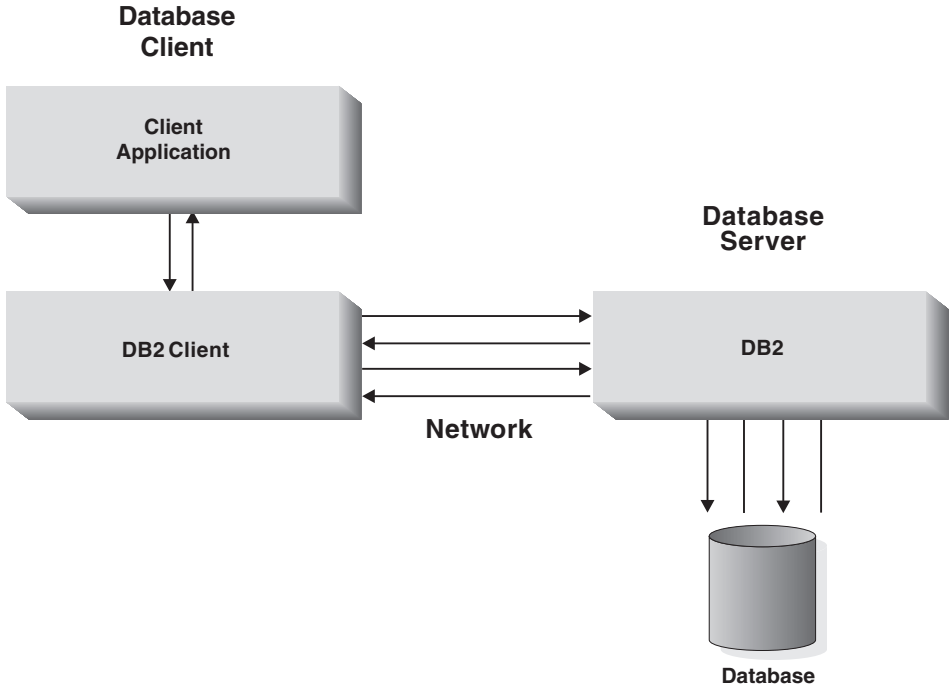


Figure 3. Application Accessing a Database on a Server

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 4 shows this feature.

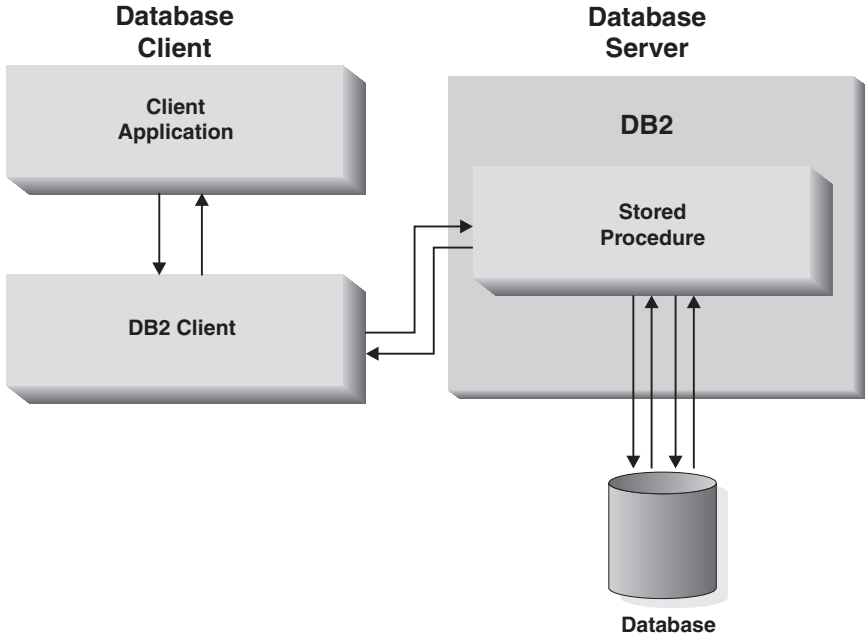


Figure 4. Application Using a Stored Procedure

Applications using stored procedures have the following advantages:

- Reduced network traffic

  A properly designed application that processes large amounts of data using stored procedures returns only the data that is needed by the client. This reduces the amount of data transmitted across the network.

- Improved performance of server intensive work

  The more SQL statements that are grouped together for execution, 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, including:
  - 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.

Writing Stored Procedures

An application design that includes a stored procedure consists of separate client and server applications. The server application, called the *stored procedure*, is contained in a shared library or class library on the server. You must compile and access the stored procedure on the server instance where the database resides. The *client application* contains a CALL statement to the stored procedure. The CALL statement can pass parameters to and return parameters from the stored procedure. You can write the stored procedure and the client application using different languages. The client application can be executed on on a platform different from the stored procedure.

The client application performs the following tasks:

1. Declares, allocates, and initializes storage for the optional data structures and host variables.
2. Connects to a database by executing the CONNECT TO statement, or by doing an implicit connect. Refer to the *SQL Reference* for details.
3. Invokes the stored procedure through the SQL CALL statement.
4. Issues a COMMIT or ROLLBACK to the database.

   Note: While the stored procedure can issue COMMIT or ROLLBACK statements, the recommended practice is to have the client application issue to issue the COMMIT or ROLLBACK. This enables your client application to evaluate the data returned by the stored procedure and to decide whether to commit the transaction or roll it back.
5. 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 tasks:

1. Accepts the parameters from the client application.
2. Executes on the database server under the same transaction as the client application.
3. Optionally, issues one or more COMMIT or ROLLBACK statements.

   Note: While the stored procedure can issue COMMIT or ROLLBACK statements, the recommended practice is to have the client application issue the COMMIT or ROLLBACK statements. This enables your client application to evaluate the data returned by the stored procedure and to decide whether to commit the transaction or roll it back.

4. 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 put several stored procedures into one library.

This chapter describes how to write stored procedures with the following parameter styles:

DB2SQL The stored procedure receives parameters that you declare in the CREATE PROCEDURE statement as host variables from the CALL statement in the client application. DB2 allocates additional parameters for DB2SQL stored procedures.

GENERAL The stored procedure receives parameters as host variables from the CALL statement in the client application. The stored procedure does not directly pass null indicators to the client application. GENERAL is the equivalent of SIMPLE stored procedures for DB2 Universal Database for OS/390.

GENERAL WITH NULLS
For each parameter declared by the user, DB2 allocates a corresponding INOUT parameter null indicator. Like GENERAL, parameters are passed as host variables. GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 Universal Database for OS/390.

JAVA The stored procedure uses a parameter passing convention that conforms to the SQLJ Routines specification. The stored procedure receives IN parameters as host variables, and receives OUT and INOUT parameters as single entry arrays.

You must register each stored procedure for the previously listed parameter styles with a CREATE PROCEDURE statement. The CREATE PROCEDURE statement specifies the procedure name, arguments, location, and parameter style of each stored procedure. These parameter styles offer increased portability and scalability of your stored procedure code across the DB2 family.

For information on using the only styles of stored procedures supported by versions of DB2 prior to DB2 Universal Database Version 6, that is, the DB2DARI and DB2GENERAL parameter styles, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

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 host variables or an SQLDA structure. The SQL CALL statement can accept a series of host variables, or an SQLDA structure. Refer to the *SQL Reference* for descriptions of the SQL CALL statement and the SQLDA structure. For information on using the SQLDA structure in a client application, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

Allocating Host Variables

Use the following steps to allocate the necessary input host variables on the client side of a stored procedure:

1. Declare enough host variables for all input variables that will be passed to the stored procedure.
2. Determine which input host variables can also be used to return values back from the stored procedure to the client.
3. Declare host variables 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 host variables as possible by using them for both input and output. This will increase the efficiency of handling multiple host variables. For example, when returning an SQLCODE to the client from the stored procedure, try to use an input host variable that is declared as an INTEGER to return the SQLCODE.

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).

Calling Stored Procedures

You can invoke a stored procedure stored at the location of the database by using the SQL CALL statement. Refer to the *SQL Reference* for a complete description of the CALL statement. Using the CALL statement is the recommended method of invoking stored procedures.

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 Procedures on the Server

The stored procedure is invoked by the SQL CALL statement and executes using data passed to it by the client application. The parameter style with which you register the stored procedure in the database manager with the CREATE PROCEDURE statement determines how the stored procedure receives data from the client application.

Registering Stored Procedures

To use the CREATE PROCEDURE statement, you must declare the following:

- Procedure name
- Mode, name, and SQL data type of each parameter
- EXTERNAL name and location
- PARAMETER STYLE

Your CREATE PROCEDURE should also declare the following:

- Whether it runs FENCED or NOT FENCED
- The type of SQL statements contained in the procedure body, if any

You can find more information on the CREATE PROCEDURE statement, including its full syntax and options for DB2 family compatibility, in the *SQL Reference*. Descriptions of typical usages of the CREATE PROCEDURE statement follow.

Procedure Names: You can overload stored procedures only by using the same name for procedures that accept a unique number of parameters. Since DB2 does not distinguish between data types, you cannot overload stored procedures based on parameter data types.

For example, issuing the following CREATE PROCEDURE statements will work because they accept one and two parameters, respectively:

```
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER, IN VAR2 INTEGER) ...
```

However, DB2 will fail to register the second stored procedure in the following example because it has the same number of parameters as the first stored procedure with the same name:

```
CREATE PROCEDURE OVERLOADFAIL (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOADFAIL (IN VAR2 VARCHAR(15)) ...
```

Parameter Modes: An *explicit parameter* is a parameter that you explicitly declare in the parameter list of the CREATE PROCEDURE statement. An *implicit parameter* is a parameter that is automatically supplied by DB2; for

example, a PARAMETER STYLE GENERAL WITH NULLS stored procedure automatically supplies an array of null indicators for the explicit parameters. When you write a stored procedure, you must consider both the explicit and implicit parameters for your stored procedure. When you write a client application, you only have to handle the explicit parameters for the stored procedure. You must declare every explicit parameter as either an IN, OUT, or INOUT parameter with a name and SQL data type. For examples of CREATE PROCEDURE statements, see "Variable Declaration and CREATE PROCEDURE Examples" on page 212.

IN Passes a value to the stored procedure from the client application, but returns no value to the client application when control returns to the client application

OUT Stores a value that is passed to the client application when the stored procedure terminates

INOUT
Passes a value to the stored procedure from the client application, and returns a value to the client application when the stored procedure terminates

Location: The EXTERNAL clause of the CREATE PROCEDURE statement tells the database manager the location of the library that contains the stored procedure. If you do not specify an absolute path for the library, or a jar name for Java stored procedures, the database manager searches the *function directory*. The *function directory* is a directory defined for your operating system as follows:

Unix operating systems
`sqllib/function`

OS/2 or Windows 32-bit operating systems
instance_name\function, where *instance_name* represents the value of the *DB2INSTPROF* instance-specific registry setting. If *DB2INSTPROF* is not set, *instance_name* represents the value of the *%DB2PATH%* environment variable. The default value of the *%DB2PATH%* environment variable is the path in which you installed DB2.

If DB2 does not find the stored procedure in *instance_name*\function, DB2 searches the directories defined by the *PATH* and *LIBPATH* environment variables.

For example, the function directory for a Windows 32-bit operating system server with DB2 installed in the C:\sqllib directory, where you have not set the *DB2INSTPROF* registry setting, is:

```
C:\sqllib\function
```

Note: You should give your library a name that is different than the stored procedure name. If DB2 locates the library in the search path, DB2 executes any stored procedure with the same name as the library which contains the stored procedure as a FENCED DB2DARI procedure.

For LANGUAGE C stored procedures, specify:

- The library name, taking the form of either:
  - A library found in the function directory
  - An absolute path including the library name
- The entry point for the stored procedure in the library. If you do not specify an entry point, the database manager will use the default entry point. The IBM XLC compiler on AIX allows you to specify any exported function name in the library as the default entry point. This is the function that is called if only the library name is specified in a stored procedure call or CREATE FUNCTION statement. To specify a default entry point, use the `-e` option in the link step. For example: `-e` *`funcname`* makes *funcname* the default entry point. On other UNIX platforms, no such mechanism exists, so the default entry point is assumed by DB2 to be the same name as the library itself.

On a UNIX-based system, for example, `mymod!proc8` directs the database manager to the `sqllib/function/mymod` library and to use entry point `proc8` within that library. On Windows 32-bit and OS/2 operating systems `mymod!proc8` directs the database manager to load the `mymod.dll` file from the function directory and call the `proc8()` procedure in the dynamic link library (DLL).

For LANGUAGE JAVA stored procedures, use the following syntax:

```
[<jar-file-name>:]<class-name>.<method-name> (java-method-signature)
```

The following list defines the EXTERNAL keywords for Java stored procedures:

jar-file-name
: If a jar file installed in the database contains the stored procedure method, you must include this value. The keyword represents the name of the jar file, and is delimitied by a colon (`:`). If you do not specify a jar file name, the database manager looks for the class in the function directory. For more information on installing jar files, see "Java Stored Procedures and UDFs" on page 668.

class-name
: The name of the class that contains the stored procedure method. If the class is part of a package, you must include the complete package name as a prefix.

method-name
The name of the stored procedure method.

java-method-signature
A list of the Java parameter data types for the method. These data types must correspond to the default Java type mapping for the signature specified after the procedure or function name. For example, the default Java mapping of the SQL type INTEGER is `int`, not `java.lang.Integer`. For a list of the default Java type mappings, see Table 32 on page 640.

For example, if you specify `MyPackage.MyClass.myMethod`, the database manager uses the `myMethod` method in the `MyClass` class, within the `MyPackage` package. DB2 recognizes that `MyPackage` refers to a package rather than a jar file because it uses a period (`.`) delimiter instead of a colon (`:`) delimiter. DB2 searches the function directory for the `MyPackage` package.

For more information on the function directory, see "Location" on page 200.

LANGUAGE: For C/C++, declare LANGUAGE C in your CREATE PROCEDURE statement. For Java stored procedures, declare LANGUAGE JAVA. For OLE stored procedures on Windows 32-bit operating systems, declare LANGUAGE OLE. For COBOL stored procedures, declare LANGUAGE COBOL. For Fortran or REXX stored procedures, you must write the stored procedure as a DB2DARI stored procedure. For more information on writing DB2DARI stored procedures, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

LANGUAGE C
The database manager calls the stored procedure using ANSI C calling and linkage conventions. Use this option for most C/C++ stored procedures.

LANGUAGE JAVA
The database manager calls the stored procedure as a method in a Java class. Use this option for any Java stored procedure.

LANGUAGE OLE
The database manager calls the stored procedure as a OLE function. Use this option for any OLE stored procedure on Windows 32-bit operating systems. Before issuing the CREATE PROCEDURE statement, you must register the DLL that contains the OLE stored procedure using the REGSVR32 command. OLE stored procedures must run in FENCED mode. For more information on using OLE stored procedures, refer to the *Application Building Guide*.

LANGUAGE COBOL

The database manager calls the stored procedure using COBOL calling and linkage conventions. Use this option for COBOL stored procedures.

Passing Parameters as Subroutines: C stored procedures of PROGRAM TYPE SUB accept arguments as subroutines. Pass numeric data type parameters as pointers. Pass character data types as arrays of the appropriate length. For example, the following C stored procedure signature accepts parameters of type INTEGER, SMALLINT, and CHAR(3):

```
int storproc (sqlint32 *arg1, short *arg2, char arg[4])
```

Java stored procedures can only accept arguments as subroutines. Pass IN parameters as simple arguments. Pass OUT and INOUT parameters as arrays with a single element. For example, the following Java stored procedure signature accepts an IN parameter of type INTEGER, an OUT parameter of type SMALLINT, and an INOUT parameter of type CHAR(3):

```
int storproc (int arg1, short arg2[], String arg[])
```

Passing Parameters as main Functions: To write a stored procedure that accepts arguments like a main function in a C program, specify PROGRAM TYPE MAIN in the CREATE PROCEDURE statement. You must write stored procedures of PROGRAM TYPE MAIN to conform to the following specifications:

- DB2 sets the value of the first element in the parameter array to the stored procedure name
- the stored procedure accepts parameters through two arguments:
  - a parameter counter variable; for example, *argc*
  - an array containing the parameters; for example, *argv[]*
- the stored procedure must be built as a shared library

In PROGRAM TYPE MAIN stored procedures, DB2 sets the value of the first element in the *argv* array, (*argv[0]*), to the name of the stored procedure. The remaining elements of the *argv* array correspond to the parameters declared in the CREATE PROCEDURE statement for the stored procedure. For example, the following embedded C stored procedure passes in one IN parameter as *argv[1]* and returns two OUT parameters as *argv[2]* and *argv[3]*.

The CREATE PROCEDURE statement for the PROGRAM TYPE MAIN example is as follows:

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
   OUT salary DOUBLE, OUT errorcode INTEGER)
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE GENERAL
```

```
    NO DBINFO
    FENCED
    READS SQL DATA
    PROGRAM TYPE MAIN
    EXTERNAL NAME 'spserver!mainexample'
```

The following code for the stored procedure copies the value of *argv[1]* into the CHAR(8) host variable *injob*, then copies the value of the DOUBLE host variable *outsalary* into *argv[2]* and returns the SQLCODE as *argv[3]*:

```
EXEC SQL BEGIN DECLARE SECTION;
  char injob[9];
  double outsalary;
EXEC SQL END DECLARE SECTION;

SQL_API_RC SQL_API_FN main_example (int argc, char **argv)
{
  EXEC SQL INCLUDE SQLCA;

  /* argv[0] contains the procedure name, so parameters start at argv[1] */
  strcpy (injob, (char *)argv[1]);

  EXEC SQL SELECT AVG(salary)
    INTO :outsalary
    FROM employee
    WHERE job = :injob;

  memcpy ((double *)argv[2], (double *)&outsalary, sizeof(double));

  memcpy ((sqlint32 *)argv[3], (sqlint32 *)&SQLCODE, sizeof(sqlint32));

  return (0);

} /* end main_example function */
```

PARAMETER STYLE: Table 9 summarizes the combinations of PARAMETER STYLE (horizontal axis) and LANGUAGE (vertical axis) allowed in CREATE PROCEDURE statements for DB2 Version 7.

Table 9. CREATE PROCEDURE: Valid Combinations of PARAMETER STYLE and LANGUAGE

	GENERAL, GENERAL WITH NULLS	JAVA	DB2SQL	DB2DARI	DB2GENERAL
LANGUAGE C	Y	N	Y	Y	N
LANGUAGE JAVA	N	Y	N	N	Y
LANGUAGE OLE	N	N	Y	N	N
LANGUAGE COBOL	Y	N	Y	N	N

GENERAL

The stored procedure receives parameters as host variables from the CALL statement in the client application. The stored procedure does not directly pass null indicators to the client application. You can only use GENERAL when you also specify the LANGUAGE C or LANGUAGE COBOL option.

DB2 Universal Database for OS/390 compatibility note: GENERAL is the equivalent of SIMPLE.

PARAMETER STYLE GENERAL stored procedures accept parameters in the manner indicated by the value of the PROGRAM TYPE clause. The following example demonstrates a PARAMETER STYLE GENERAL stored procedure that accepts two parameters using PROGRAM TYPE SUBROUTINE:

```
SQL_API_RC SQL_API_FN one_result_set_to_client
   (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  EXEC SQL BEGIN DECLARE SECTION;
    double l_insalary;
  EXEC SQL END DECLARE SECTION;

  l_insalary = *insalary;
  *out_sqlerror = 0;

  EXEC SQL DECLARE c3 CURSOR FOR
    SELECT name, job, CAST(salary AS INTEGER)
    FROM staff
    WHERE salary > :l_insalary
    ORDER BY salary;

  EXEC SQL OPEN c3;
  /* Leave cursor open to return result set */

  return (0);

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
  return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end one_result_set_to_client function */
```

GENERAL WITH NULLS

For each parameter declared by the user, DB2 allocates a corresponding INOUT parameter null indicator. Like GENERAL,

parameters are passed as host variables. You can only use GENERAL WITH NULLS when you also specify the LANGUAGE C or LANGUAGE COBOL option.

DB2 Universal Database for OS/390 compatibility note: GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS.

PARAMETER STYLE GENERAL WITH NULLS stored procedures accept parameters in the manner indicated by the value of the PROGRAM TYPE clause, and allocate an array of null indicators with one element per declared parameter. The following SQL registers a PARAMETER STYLE GENERAL WITH NULLS stored procedure that passes one INOUT parameter and two OUT parameters using PROGRAM TYPE SUB:

```
CREATE PROCEDURE INOUT_PARAM (INOUT medianSalary DOUBLE,
   OUT errorCode INTEGER, OUT errorLabel CHAR(32))
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE GENERAL WITH NULLS
   NO DBINFO
   FENCED
   MODIFIES SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!inout_param'
```

The following C code demonstrates how to declare and use the null indicators required by a GENERAL WITH NULLS stored procedure:

```
SQL_API_RC SQL_API_FN inout_param (double *inoutMedian,
  sqlint32 *out_sqlerror, char buffer[33], sqlint16 nullinds[3])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
    nullinds[0] = -1;
    nullinds[1] = -1;
    nullinds[2] = -1;
  }
  else
  {
    int counter = 0;
    *out_sqlerror = 0;
    medianSalary = *inoutMedian;

    strcpy(buffer, "DECLARE inout CURSOR");
    EXEC SQL DECLARE inout CURSOR FOR
      SELECT CAST(salary AS DOUBLE) FROM staff
      WHERE salary > :medianSalary
      ORDER BY salary;
```

```
  nullinds[1] = 0;
  nullinds[2] = 0;

  strcpy(buffer, "SELECT COUNT INTO numRecords");
  EXEC SQL SELECT COUNT(*) INTO :numRecords
    FROM staff
    WHERE salary > :medianSalary;

  if (numRecords != 0)
  /* At least one record was found */
  {
    strcpy(buffer, "OPEN inout");
    EXEC SQL OPEN inout USING :medianSalary;

    strcpy(buffer, "FETCH inout");
    while (counter < (numRecords / 2 + 1)) {
      EXEC SQL FETCH inout INTO :medianSalary;

      *inoutMedian = medianSalary;
      counter = counter + 1;
    }

    strcpy(buffer, "CLOSE inout");
    EXEC SQL CLOSE inout;
  }
  else /* No records were found */
  {
    /* Return 100 to indicate NOT FOUND error */
    *out_sqlerror = 100;
  }
}

return (0);

/* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
{
  *out_sqlerror = SQLCODE;
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  return (0);
}

} /* end inout_param function */
```

JAVA The stored procedure uses a parameter passing convention that conforms to the *SQLJ Routines* specification. The stored procedure receives IN parameters as host variables, and receives OUT and INOUT parameters as single entry arrays. You can only use JAVA when you also specify the LANGUAGE JAVA option.

DB2SQL

Your C function definition for a DB2SQL stored procedure must

append the following implicit parameters to the definition for the parameters declared in the CREATE PROCEDURE statement:

```
sqlint16 nullinds[n],  1
char sqlst[6],         2
char qualname[28],     3
char specname[19],     4
char diagmsg[71],      5
```

DB2 passes the following arguments to the stored procedure:

1. DB2 allocates an array of implicit SMALLINT INOUT parameters as null indicators for the explicit parameters. The array is of size *n*, where *n* represents the number of explicit parameters.
2. An implicit CHAR(5) OUT parameter for an SQLSTATE value.
3. An implicit CHAR(27) IN parameter for the qualified stored procedure name.
4. An implicit CHAR(18) IN parameter for the specific name of the stored procedure.
5. An implicit CHAR(70) OUT parameter for an SQL diagnostic string.

You can only specify DB2SQL when you also specify the LANGUAGE C or LANGUAGE COBOL option. For example, the following CREATE PROCEDURE statement registers a PARAMETER STYLE DB2SQL stored procedure:

```
CREATE PROCEDURE DB2SQL_EXAMPLE (IN job CHAR(8), OUT salary DOUBLE)
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE DB2SQL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!db2sqlexample'
```

Write the stored procedure using the following conventions:

- PARAMETER STYLE DB2SQL stored procedures pass an array of null indicators with one element for each explicit parameter. A negative value of the null indicator element for an IN or INOUT parameter indicates that the client application passed in a null value for that parameter. To indicate that an output parameter is not NULL, set the value of the null indicator element for the OUT or INOUT parameter to 0. To indicate that an output parameter is NULL, set the value of the null indicator element for the OUT or INOUT parameter to -1.
- Append the arguments in the stored procedure signature for the DB2SQL parameters, as previously described.

- You can set the value of the DB2SQL SQLSTATE (CHAR(5) and diagnostic message (null-terminated CHAR(70)) parameters to return a customized value in the SQLCA to the client.

For example, the following embedded C stored procedure demonstrates the coding style for PARAMETER STYLE DB2SQL stored procedures:

```
SQL_API_RC SQL_API_FN db2sql_example (
  char injob[9],         /* Input - CHAR(8)   */
  double *salary,        /* Output - DOUBLE   */
  sqlint16 nullinds[2],
  char sqlst[6],
  char qualname[28],
  char specname[19],
  char diagmsg[71]
)
{
  EXEC SQL INCLUDE SQLCA;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
    nullinds[1] = -1;
    /* Set custom SQLSTATE to return to client. */
    strcpy(sqlst, "38100");
    /* Set custom message to return to client. */
    strcpy(diagmsg, "Received null input on call to DB2SQL_EXAMPLE.");

  }
  else
  {
    EXEC SQL SELECT (CAST(AVG(salary) AS DOUBLE))
      INTO :outsalary INDICATOR :outsalaryind
      FROM employee
      WHERE job = :injob;

    *salary = outsalary;
    nullinds[1] = outsalaryind;
  }
  return (0);
} /* end db2sql_example function */
```

The following embedded C client application demonstrates how to issue a CALL statement that invokes the DB2SQL_EXAMPLE stored procedure. Note that the example includes null indicators for each parameter in the CALL statement. The example sets the null indicator *in_jobind* to 0 to indicate that a non-NULL value is being passed to the stored procedure for the IN parameter represented by the host variable *in_job*. The null indicators for the OUT parameters are set to -1 to indicate that no input is being passed to the stored procedure for those parameters.

```
int db2sqlparm(char out_lang[9], char job_name[9])
{
   int testlang;

   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to DB2SQL_EXAMPLE */
     char in_job[9];
     sqlint16 in_jobind;
     double out_salary = 0;
     sqlint16 out_salaryind;
   EXEC SQL END DECLARE SECTION;

   /*******************************************************\
   * Call DB2SQL_EXAMPLE stored procedure                  *
   \*******************************************************/

   testlang = strncmp(out_lang, "C", 1);
   if (testlang != 0) {
      /* Only LANGUAGE C procedures can be PARAMETER STYLE DB2SQL,
         so do not call the DB2SQL_EXAMPLE stored procedure */
      printf("\nStored procedures are not implemented in C.\n"
             "Skipping the call to DB2SQL_EXAMPLE.\n");
   }
   else {
      strcpy(procname, "DB2SQL_EXAMPLE");
      printf("\nCALL stored procedure named %s\n", procname);

      /* out_salary is an OUT parameter, so set the
         null indicator to -1 to indicate no input value */
      out_salaryind = -1;

      strcpy(in_job, job_name);

      /* in_job is an IN parameter, so check to
          see if there is any input value */
      if (strlen(in_job) == 0)
      {
         /* in_job is null, so set the null indicator
            to -1 to indicate there is no input value */
         in_jobind = -1;
         printf("with NULL input, to return a custom
                 SQLSTATE and diagnostic message\n");
      }
      else
      {
         /* in_job is not null, so set the null indicator
            to 0 to indicate there is an input value */
         in_jobind = 0;
      }

      /* DB2SQL_EXAMPLE is PS DB2SQL, so pass
         a null indicator for each parameter */
      EXEC SQL CALL :procname (:in_job:in_jobind,
          :out_salary:out_salaryind);
```

```
      /* DB2SQL stored procedures can return a custom
         SQLSTATE and diagnostic message, so instead of
         using the EMB_SQL_CHECK macro to check the value
         of the returned SQLCODE, check the SQLCA structure for
         the value of the SQLSTATE and the diagnostic message */

      /* Check value of returned SQLSTATE */
      if (strncmp(sqlca.sqlstate, "00000", 5) == 0) {
        printf("Stored procedure returned successfully.\n");
        printf("Average salary for job %s = %9.2f\n",
          in_job, out_salary);
      }
      else {
         printf("Stored procedure failed with SQLSTATE %s.\n",
           sqlca.sqlstate);
         printf("Stored procedure returned the following
           diagnostic message:\n");
         printf("  \"%s\"\n", sqlca.sqlerrmc);
      }
   }

  return 0;
}
```

DB2GENERAL
The stored procedure uses a parameter passing convention that is only supported by DB2 Java stored procedures. You can only use DB2GENERAL when you also specify the LANGUAGE JAVA option.

For increased portability, you should write Java stored procedures using the PARAMETER STYLE JAVA conventions. See "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765 for more information on writing DB2GENERAL parameter style stored procedures.

DB2DARI
The stored procedure uses a parameter passing convention that conforms with C language calling and linkage conventions. This option is only supported by DB2 Universal Database, and can only be used when you also specify the LANGUAGE C option.

To increase portability across the DB2 family, you should write your LANGUAGE C stored procedures using the GENERAL or GENERAL WITH NULLS parameter styles. If you want to write DB2DARI parameter style stored procedures, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

Passing a DBINFO Structure: For LANGUAGE C stored procedures with a PARAMETER TYPE of GENERAL, GENERAL WITH NULLS, or DB2SQL, you have the option of writing your stored procedure to accept an additional parameter. You can specify DBINFO in the CREATE PROCEDURE statement to instruct the client application to pass a DBINFO structure containing

information about the DB2 client to the stored procedure, along with the call parameters. The DBINFO structure contains the following values:

Database name
The name of the database to which the client is connected.

Application authorization ID
The application run-time authorization ID.

Code page
The code page of the database.

Schema name
Not applicable to stored procedures.

Table name
Not applicable to stored procedures.

Column name
Not applicable to stored procedures.

Database version and release
The version, release, and modification level of the database server invoking the stored procedure.

Platform
The platform of the database server.

Table function result column numbers
Not applicable to stored procedures.

For more information on the DBINFO structure, see "DBINFO Structure" on page 404.

Variable Declaration and CREATE PROCEDURE Examples

The following examples demonstrate the stored procedure source code and CREATE PROCEDURE statements you would use in hypothetical scenarios with the SAMPLE database.

Using IN and OUT Parameters: Assume that you want to create a Java stored procedure `GET_LASTNAME` that, given *empno* (SQL type VARCHAR), returns *lastname* (SQL type CHAR) from the EMPLOYEE table in the SAMPLE database. You will create the procedure as the `getname` method of the Java class `StoredProcedure`, contained in the JAR installed as `myJar`. Finally, you will call the stored procedure with a client application coded in C.

1. Declare two host variables in your stored procedure source code:

```
String empid;
String name;
...
#sql { SELECT lastname INTO :empid FROM employee WHERE empno=:empid }
```

2. Register the stored procedure with the following CREATE PROCEDURE statement:

```
CREATE PROCEDURE GET_LASTNAME (IN EMPID CHAR(6), OUT NAME VARCHAR(15))
    EXTERNAL NAME 'myJar:StoredProcedure.getname'
    LANGUAGE JAVA PARAMETER STYLE JAVA FENCED
    READS SQL DATA
```

3. Call the stored procedure from your client application written in C:

```
EXEC SQL BEGIN DECLARE SECTION;
   struct name { short int; char[15] }
   char[7] empid;
EXEC SQL END DECLARE SECTION;
   ...
EXEC SQL CALL GET_LASTNAME (:empid, :name);
```

Using INOUT Parameters: For the following example, assume that you want to create a C stored procedure *GET_MANAGER* that, given *deptnumb* (SQL type SMALLINT), returns *manager* (SQL type SMALLINT) from the ORG table in the SAMPLE database.

1. Since *deptnumb* and *manager* are both of SQL data type SMALLINT, you can declare a single variable *onevar* in your stored procedure that receives a value from and returns a value to the client application:

```
EXEC SQL BEGIN DECLARE SECTION;
  short onevar = 0;
EXEC SQL END DECLARE SECTION;
```

2. Register the stored procedure with the following CREATE PROCEDURE statement:

```
CREATE PROCEDURE GET_MANAGER (INOUT onevar SMALLINT)
   EXTERNAL NAME 'stplib!getman'
   LANGUAGE C PARAMETER STYLE GENERAL FENCED
   READS SQL DATA
```

3. Call the stored procedure from your client application written in Java:

```
short onevar = 0;
  ...
#SQL { CALL GET_MANAGER (:INOUT onevar) };
```

SQL Statements in Stored Procedures

Stored procedures can contain SQL statements. When you issue the CREATE PROCEDURE statement, you should specify the type of SQL statements the stored procedure contains, if any. If you do not specify a value when you register the stored procedure, the database manager uses MODIFIES SQL DATA. To restrict the type of SQL used in the stored procedure, you can use one of the following four options:

NO SQL

Indicates that the stored procedure cannot execute any SQL statements. If the stored procedure attempts to execute an SQL statement, the statement returns SQLSTATE 38001.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure. If the stored procedure attempts to execute an SQL statement that reads or modifies SQL data, the statement returns SQLSTATE 38004. Statements that are not supported in any stored procedure return SQLSTATE 38003.

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be executed by the stored procedure. If the stored procedure attempts to execute an SQL statement that modifies data, the statement returns SQLSTATE 38002. Statements that are not supported in any stored procedure return SQLSTATE 38003.

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures. If the stored procedure attempts to execute an SQL statement that is not supported in a stored procedure, the statement returns SQLSTATE 38003.

For more information on the CREATE PROCEDURE statement, refer to the *SQL Reference*.

Nested Stored Procedures

Nested stored procedures are stored procedures that call another stored procedure. You can use this technique in your DB2 applications under the following restrictions:

- the stored procedures must be cataloged as LANGUAGE C or LANGUAGE SQL.
- the calling stored procedure can only call a stored procedure that is cataloged using the same LANGUAGE clause. For nested calls only, LANGUAGE C and LANGUAGE SQL are considered the same language. For example, a LANGUAGE C stored procedure can call an SQL procedure.
- the calling stored procedure cannot call a stored procedure that is cataloged with a higher SQL data access level. For example, a stored procedure cataloged with CONTAINS SQL data access can call a stored procedure cataloged with NO SQL or CONTAINS SQL data access, but cannot call a stored procedure cataloged with READS SQL DATA or MODIFIES SQL DATA.
- up to 16 levels of nested stored procedure calls are supported. For example, a scenario where stored procedure PROC1 calls PROC2, and PROC2 calls PROC3 represents three levels of nested stored procedures.
- the calling and called stored procedures at all levels of nesting cannot be cataloged as NOT FENCED

Nested SQL procedures can return one or more result sets to the client application or to the calling stored procedure. To return a result set from an SQL procedure to the client application, issue the DECLARE CURSOR statement using the WITH RETURN TO CLIENT clause. To return a result set from an SQL procedure to the caller, where the caller is either a client application or a calling stored procedure, issue the DECLARE CURSOR statement using the WITH RETURN TO CALLER clause.

Nested embedded SQL stored procedures written in C and nested CLI stored procedures cannot return result sets to the client application or calling stored procedure. If a nested embedded SQL stored procedure or a nested CLI stored procedure leaves cursors open when the stored procedure exits, DB2 closes the cursors. For more information on returning result sets from stored procedures, see "Returning Result Sets from Stored Procedures" on page 233.

Using Cursors in Recursive Stored Procedures

To avoid errors when using SQL procedures or stored procedures written in embedded SQL, close all open cursors before issuing a recursive CALL statement.

For example, assume the stored procedure MYPROC contains the following code fragment:

```
OPEN c1;
CALL MYPROC();
CLOSE c1;
```

DB2 returns an error when MYPROC is called because cursor c1 is still open when MYPROC issues a recursive CALL statement. The specific error returned by DB2 depends on the actions MYPROC performs on the cursor.

To successfully call MYPROC, rewrite MYPROC to close any open cursors *before* the nested CALL statement as shown in the following example:

```
OPEN c1;
CLOSE c1;
CALL MYPROC();
```

Close all open cursors before issuing the nested CALL statement to avoid an error.

Restrictions

When you create a stored procedure, you must observe the following restrictions:

- Do not use the standard I/O streams, for example, calls to `System.out.println()` in Java, `printf()` in C/C++, or `display` in COBOL. Stored procedures run in the background, so you cannot write to the screen. However, you can write to a file.

- Include only the SQL statements allowed by the CREATE PROCEDURE statement with which you register the stored procedure. For information on using the NO SQL, READS SQL DATA, CONTAINS SQL, or MODIFIES SQL DATA clauses to catalog your stored procedure, see "SQL Statements in Stored Procedures" on page 213.
- You cannot use COMMIT statements in stored procedures when either or both of the following conditions is true:
  - you catalog the stored procedure using the NO SQL clause
  - the stored procedure is called from an application performing a multisite update
- You cannot execute any connection-related statements or commands in stored procedures, including:
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- On UNIX-based systems, NOT FENCED stored procedures run under the user ID of the DB2 Agent Process. FENCED stored procedures run under the user ID of the `db2dari` executable, which is set to the owner of the .fenced file in sqllib/adm. This user ID controls the system resources available to the stored procedure. For information on the `db2dari` executable, refer to the *Quick Beginnings* book for your platform.
- You cannot overload stored procedures that accept the same number of parameters, even if the parameters are of different SQL data types.
- 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.

Writing OLE Automation Stored Procedures

OLE (Object Linking and Embedding) automation is part of the OLE 2.0 architecture from Microsoft Corporation. DB2 can invoke methods of OLE automation objects as external stored procedures. For an overview of OLE automation, see "Writing OLE Automation UDFs" on page 425.

After you code an OLE automation object, you must register the methods of the object as stored procedures using the CREATE PROCEDURE statement. To register an OLE automation stored procedure, issue a CREATE PROCEDURE statement with 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). The OLE automation object must be implemented as an in-process server (.DLL).

The following CREATE PROCEDURE statement registers an OLE automation stored procedure called “median” for the “median” method of the OLE automation object “db2smpl.salary”:

```
CREATE PROCEDURE median (INOUT sal DOUBLE)
   EXTERNAL NAME 'db2smpl.salary!median'
   LANGUAGE OLE
   FENCED
   PARAMETER STYLE DB2SQL
```

The calling conventions for OLE method implementations are identical to the conventions for procedures written in C or C++.

DB2 automatically handles the type conversions between SQL types and OLE automation types. For a list of the DB2 mappings between supported OLE automation types and SQL types, see Table 16 on page 427. For a list of the DB2 mappings between SQL types and the data types of the OLE programming language, such as BASIC or C/C++, see Table 17 on page 428.

Data passed between DB2 and OLE automation stored procedures is passed as call by reference. DB2 does not support SQL types such as DECIMAL or LOCATORS, or OLE automation types such as boolean or CURRENCY, that are not listed in the previously referenced tables. Character and graphic data mapped to BSTR is converted from the database code page to 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).

After you code an OLE automation object, you must register the methods of the object as stored procedures using the CREATE PROCEDURE statement. To register an OLE automation stored procedure, issue a CREATE PROCEDURE statement with 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). The OLE automation object needs to be implemented as an in-process server (.DLL).

Example OUT Parameter Stored Procedure

Following is a sample program demonstrating the use of an OUT host variable. 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 OUT host variable.

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 $n \div 2 + 1$ 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 5.

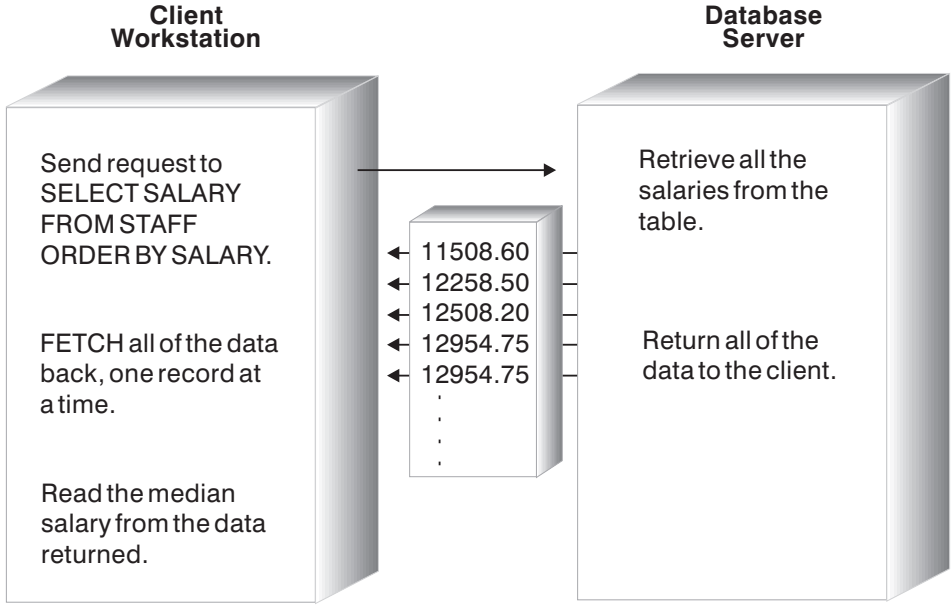


Figure 5. Median Sample Without a Stored Procedure

Since only the salary at row $n \div 2 + 1$ is needed, the application discards all the additional data, *but only after it is transmitted across the network.*

You can design an application using the stored procedures technique that allows the stored procedure to process and discard the unnecessary data, returning only the median salary to the client application. Figure 6 shows this feature.

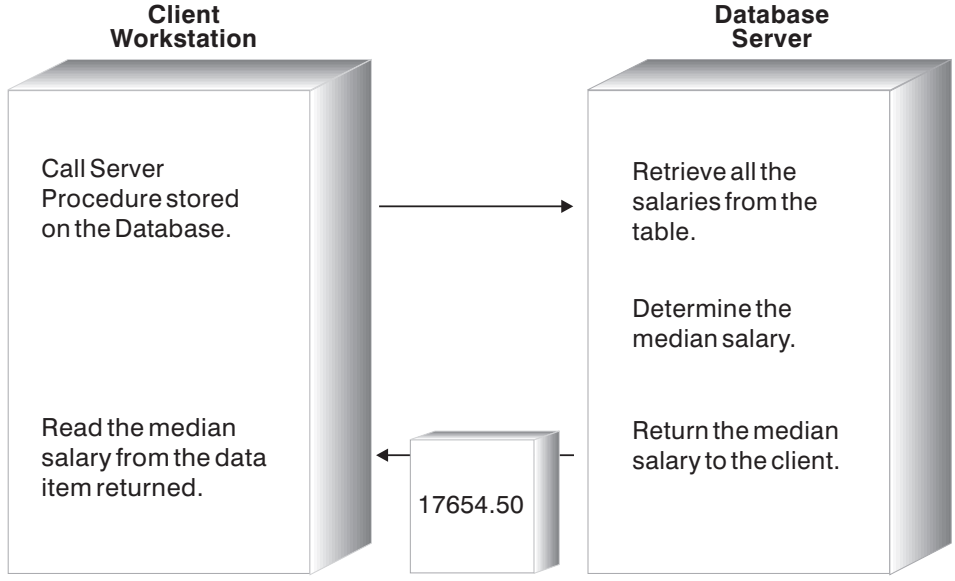


Figure 6. OUT Parameter Sample Using a Stored Procedure

"OUT Client Description" shows a sample OUT host variable client application and stored procedure. The sample programs are available in Java as:

Client application	Outcli.java
Stored procedure	Outsrv.sqlj

The sample programs are available in C as:

Client application	spclient.sqc
Stored procedure	spserver.sqc

OUT Client Description

1. **Include Files**. The C client applications include the following files:

SQL	Defines the symbol SQL_TYP_FLOAT
SQLDA	Defines the descriptor area
SQLCA	Defines the communication area for error handling

   The JDBC client application imports the following packages:

java.sql.*	JDBC classes from the Java implementation on your client
java.math.BigDecimal	Provides Java support for the DB2 DECIMAL data type

2. **Connect to Database**. The application must connect to the database before calling the stored procedure.
3. **Turn off Autocommit.** The client application explicitly disables autocommit before calling the stored procedure. Disabling autocommit allows the client application to control whether the work performed by the stored procedure control is rolled back or committed. The stored procedure for this example returns an OUT parameter containing an SQLCODE value so that client applications can easily use condition statements to commit or roll back the work performed by the stored procedure.
4. **Declare and Initialize the Host Variable**. This step declares and initializes the host variable. Java programs must register the data type of each INOUT or OUT parameter and initialize the value of every parameter before invoking the stored procedure.
5. **Call the Stored Procedure**. The client application calls the stored procedure `OUTPARAM` for the database SAMPLE using a CALL statement with three parameters.
6. **Retrieve the Output Parameters**. JDBC client applications must explicitly retrieve the values of the output parameters returned by the stored procedure. For C/C++ client applications, DB2 updates the value of the host variables used in the CALL statement when the client application executes the CALL statement.
7. **Check the Value of the Returned SQLCODE**. The client application checks the value of the OUT parameter containing the SQLCODE to determine whether to roll back or commit the transaction.
8. **Disconnect from Database**. To help DB2 free system resources held for each connection, you should explicitly close the connection to the database before exiting the client application.

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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

Java Any SQL error is thrown as an `SQLException` and handled in the `catch` block of the application.

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

Example OUT Client Application: Java

```
import java.sql.*;            // JDBC classes                                  1
import java.math.BigDecimal;  // BigDecimal support for packed decimal type

class Spclient
{
  static String sql = "";
  static String procName = "";
  static String inLanguage = "";
  static CallableStatement callStmt;
  static int outErrorCode = 0;
  static String outErrorLabel = "";
  static double outMedian = 0;

  static
  {
    try
    {
      System.out.println();
      System.out.println("Java Stored Procedure Sample");
      Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
    }
    catch (Exception e)
    {
      System.out.println("\nError loading DB2 Driver...\n");
      e.printStackTrace();
    }
  }

  public static void main(String argv[])
  {
    Connection con = null;
    // URL is jdbc:db2:dbname
    String url = "jdbc:db2:sample";

    try
    {
      // connect to sample database
      // connect with default id/password
      con = DriverManager.getConnection(url);                                  2

      // turn off autocommit
      con.setAutoCommit(false);                                                3

      outLanguage(con);
      outParameter(con);
      inParameters(con);
      inoutParam(con, outMedian);
      resultSet(con);
      twoResultSets(con);
      allDataTypes(con);

      // rollback any changes to the database
      con.rollback();                                                          8
      con.close();
```

```
    }
    catch (Exception e)
    {
      try { con.close(); } catch (Exception x) { }
      e.printStackTrace ();
    }
  } // end main

  public static void outParameter(Connection con)
    throws SQLException
  {
    // prepare the CALL statement for OUT_PARAM
    procName = "OUT_PARAM";
    sql = "CALL " + procName + "(?, ?, ?)";
    callStmt = con.prepareCall(sql);

    // register the output parameter                                    4
    callStmt.registerOutParameter (1, Types.DOUBLE);
    callStmt.registerOutParameter (2, Types.INTEGER);
    callStmt.registerOutParameter (3, Types.CHAR);

    // call the stored procedure                                        5
    System.out.println ("\nCall stored procedure named " + procName);
    callStmt.execute();

    // retrieve output parameters                                       6
    outMedian = callStmt.getDouble(1);
    outErrorCode = callStmt.getInt(2);
    outErrorLabel = callStmt.getString(3);

    if (outErrorCode == 0) {                                            7
      System.out.println(procName + " completed successfully");
      System.out.println ("Median salary returned from OUT_PARAM = "
        + outMedian);
    }
    else  { // stored procedure failed
      System.out.println(procName + " failed with SQLCODE "
        + outErrorCode);
      System.out.println(procName + " failed at " + outErrorLabel);
    }
  }
}
```

Example OUT Client Application: C

```
#include <stdio.h>                                                    1
#include <stdlib.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
   /* Declare host variable for stored procedure name */
   char procname[254];

   /* Declare host variables for stored procedure error handling */
   sqlint32 out_sqlcode;                                              4
   char out_buffer[33];
EXEC SQL END DECLARE SECTION;

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

   EXEC SQL CONNECT TO sample;                                        2
   EMB_SQL_CHECK("CONNECT TO SAMPLE");

   outparameter();

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf("\nStored procedure rolled back.\n\n");

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;                                            8
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}

int outparameter() {
   /*********************************************************\
   * Call OUT_PARAM stored procedure                         *
   \*********************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to OUT_PARAM */
     double out_median;
   EXEC SQL END DECLARE SECTION;

   strcpy(procname, "OUT_PARAM");
   printf("\nCALL stored procedure named %s\n", procname);

   /* OUT_PARAM is PS GENERAL, so do not pass a null indicator */
   EXEC SQL CALL :procname (:out_median, :out_sqlcode, :out_buffer);  5  6
   EMB_SQL_CHECK("CALL OUT_PARAM");
   /* Check that the stored procedure executed successfully */
   if (out_sqlcode == 0)                                              7
   {
```

```
    printf("Stored procedure returned successfully.\n");

    /*********************************************************\
     * Display the median salary returned as an output parameter *
     \*********************************************************/

    printf("Median salary returned from OUT_PARAM = %8.2f\n", out_median);

  }
  else
  { /* print the error message, roll back the transaction */
    printf("Stored procedure returned SQLCODE %d\n", out_sqlcode);
    printf("from procedure section labelled \"%s\".\n", out_buffer);
  }

  return 0;
}
```

OUT Stored Procedure Description

1. **Declare Signature**. The procedure returns three parameters: a DOUBLE for the median value; an INTEGER for the SQLCODE, and a CHAR for any error message. You must specify the equivalent data types as arguments in the stored procedure function definition using the DB2 type mappings specified in the programming chapter for each language.
2. **Declare a CURSOR Ordered by Salary**. To work with multiple rows of data, C stored procedures issue a DECLARE CURSOR statement and JDBC stored procedures create a ResultSet object. The ORDER BY SALARY clause enables the stored procedure to retrieve salaries in an ascending order.
3. **Determine Total Number of Employees**. The stored procedure uses a simple SELECT statement with the COUNT function to retrieve the number of employees in the EMPLOYEE table.
4. **FETCH Median Salary**. The stored procedure issues successive FETCH statements until it assigns the median salary to a variable.
5. **Assign the Median Salary to the Output Variable**. To return the value of the median salary to the client application, assign the value to the argument in the stored procedure function or method declaration that corresponds to the OUT parameter.
6. **Return to the Client Application**. Only PARAMETER STYLE DB2DARI stored procedures return values to the client. For more information on DB2DARI stored procedures, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

Example OUT Parameter Stored Procedure: Java

```
import java.sql.*;              // JDBC classes
import COM.ibm.db2.jdbc.app.*;  // DB2 JDBC classes
import java.math.BigDecimal;  // Packed Decimal class

public class Spserver
{
   public static void outParameter (double[] medianSalary,
         int[] errorCode, String[] errorLabel) throws SQLException          1
   {   try
      {
         int numRecords;
         int counter = 0;
         errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

         // Get caller's connection to the database
         Connection con = DriverManager.getConnection("jdbc:default:connection");
         errorLabel[0] = "GET CONNECTION";

         String query = "SELECT COUNT(*) FROM staff";
         errorLabel[0] = "PREPARE COUNT STATEMENT";
         PreparedStatement stmt = con.prepareStatement(query);
         errorLabel[0] = "GET COUNT RESULT SET";
         ResultSet rs = stmt.executeQuery();

         // move to first row of result set
         rs.next();

         // set value for the output parameter
         errorLabel[0] = "GET NUMBER OF RECORDS";
         numRecords = rs.getInt(1);                                         3

         // clean up first result set
         rs.close();
         stmt.close();

         // get salary result set
         query = "SELECT CAST(salary AS DOUBLE) FROM staff "
            + "ORDER BY salary";
         errorLabel[0] = "PREPARE SALARY STATEMENT";
         PreparedStatement stmt2 = con.prepareStatement(query);
         errorLabel[0] = "GET SALARY RESULT SET";
         ResultSet rs2 = stmt2.executeQuery();                              2

         while (counter < (numRecords / 2 + 1))
         {
            errorLabel[0] = "MOVE TO NEXT ROW";
            rs2.next();                                                     4
            counter++;
         }
         errorLabel[0] = "GET MEDIAN SALARY";
         medianSalary[0] = rs2.getDouble(1);                                5

         // clean up resources
         rs2.close();
```

```
            stmt2.close();
            con.close();                                                    6

         }
         catch (SQLException sqle)
         {
            errorCode[0] = sqle.getErrorCode();
         }
      }
   }
```

Example OUT Parameter Stored Procedure: C

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

/* Declare function prototypes for this stored procedure library */

SQL_API_RC SQL_API_FN out_param (double *, sqlint32 *, char *);          1

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
/* Declare host variables for basic error-handling */
  sqlint32 out_sqlcode;
  char buffer[33];

/* Declare host variables used by multiple stored procedures */
  sqlint16 numRecords;
  double medianSalary;
EXEC SQL END DECLARE SECTION;

SQL_API_RC SQL_API_FN out_param (double *outMedianSalary,
  sqlint32 *out_sqlerror, char buffer[33])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  int counter = 0;
  *out_sqlerror = 0;

  strcpy(buffer, "DECLARE c1");
  EXEC SQL DECLARE c1 CURSOR FOR                                         2
    SELECT CAST(salary AS DOUBLE) FROM staff
    ORDER BY salary;

  strcpy(buffer, "SELECT");
  EXEC SQL SELECT COUNT(*) INTO :numRecords FROM staff;                  3

  strcpy(buffer, "OPEN");
  EXEC SQL OPEN c1;

  strcpy(buffer, "FETCH");
  while (counter < (numRecords / 2 + 1)) {
    EXEC SQL FETCH c1 INTO :medianSalary;                                4

    /* Set value of OUT parameter to host variable */
    *outMedianSalary = medianSalary;                                     5
    counter = counter + 1;
  }
```

```
  strcpy(buffer, "CLOSE c1");
  EXEC SQL CLOSE c1;

  return (0);                                                          6

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end out_param function */
```

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.
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.

Note: 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 Universal Database for OS/390 or DB2 Universal Database 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.)

For more information on this topic, see "Conversion Between Different Code Pages" on page 515.

C++ Consideration

When writing a stored procedure in C++, you may want to consider declaring the procedure name using `extern "C"`, as in the following example:

```
extern "C" SQL_API_RC SQL_API_FN proc_name( short *parm1, char *parm2)
```

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 decorations for the function name when you call the stored procedure.

Graphic Host Variable Considerations

Any stored procedure written in C or C++, that receives or returns graphic data through its parameter input or output should generally be precompiled with the WCHARTYPE NOCONVERT option. This is because graphic data

passed through these parameters 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 parameters 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 in C and C++" on page 621. For information on EUC code sets and application guidelines, see "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 521, and more specifically to "Considerations for Stored Procedures" on page 525.

Multisite Update Consideration

Stored procedures that applications call with CONNECT TYPE 2 cannot issue a COMMIT or ROLLBACK, either dynamically or statically.

Improving Stored Procedure Performance

To improve the performance of your stored procedures, consider implementing one or more of the following techniques:

- Use VARCHAR parameters instead of CHAR parameters, as described in "Using VARCHAR Parameters Instead of CHAR Parameters" on page 231.
- Set the DB2_STPROC_LOOKUP_FIRST registry variable to ON, as described in "Forcing DB2 to Look Up Stored Procedures in the System Catalogs" on page 231.
- Catalog the stored procedure as a NOT FENCED stored procedure, as described in "NOT FENCED Stored Procedures" on page 231.

Using VARCHAR Parameters Instead of CHAR Parameters

You can improve the performance of your stored procedures by using VARCHAR parameters instead of CHAR parameters. Using VARCHAR data types instead of CHAR data types prevents DB2 from padding parameters with spaces before passing the parameter, and decreases the amount of time required to transmit the parameter across a network.

For example, if your client application passes the string `"A SHORT STRING"` as a CHAR(200) parameter to a stored procedure, DB2 has to pad the parameter with 186 spaces, null-terminate the string, then send the entire 200 character string and null-terminator across the network to the stored procedure.

In comparison, passing the string ″A SHORT STRING″ as a VARCHAR(200) parameter to a stored procedure results in DB2 simply passing the 14 character string across the network.

Forcing DB2 to Look Up Stored Procedures in the System Catalogs

When you call a stored procedure, the default behavior for DB2 is to search the sqllib/function and sqllib/function/unfenced directories for a shared library with the same name as the stored procedure before looking up the name of the shared library for the stored procedure in the system catalog. Only stored procedures of PARAMETER TYPE DB2DARI can have the same name as their shared library, so only DB2DARI stored procedures benefit from the default behavior of DB2. If you use stored procedures cataloged with a different PARAMETER TYPE, the time that DB2 spends searching the above directories degrades the performance of those stored procedures.

To enhance the performance of stored procedures that are not cataloged as PARAMETER TYPE DB2DARI, set the value of the DB2_STPROC_LOOKUP_FIRST registry variable to ON. This registry variable forces DB2 to look up the name of the shared library for the stored procedure in the system catalog before searching the above directories.

To set the value of the DB2_STPROC_LOOKUP_FIRST registry variable to ON, issue the following command from the CLP:

```
db2set DB2_STPROC_LOOKUP_FIRST=ON
```

NOT FENCED Stored Procedures

Your stored procedure can run as either a *FENCED* or a *NOT FENCED* stored procedure, depending on whether you register the stored procedure as FENCED or NOT FENCED in the CREATE PROCEDURE statement.

A *NOT FENCED* stored procedure runs in the same address space as the database manager (the DB2 Agent's address space). Running your stored procedure as NOT FENCED results in increased performance when compared

with running it as FENCED because FENCED stored procedures, by default, run in a special DB2 process. The address space of this process is distinct from the DB2 System Controller.

Notes:

1. While you can expect performance improvements from running NOT FENCED stored procedures, user code can accidentally or maliciously damage the database control structures. You should only use NOT FENCED stored procedures when you need to maximize the performance benefits. Test all your stored procedures thoroughly prior to running them as NOT FENCED.
2. 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.

For debugging purposes, consider using *local FENCED stored procedures*. A local FENCED procedure is a PARAMETER STYLE DB2DARI procedure. To call a local FENCED procedure, issue CALL <*library-name*>!<*entry-point*>, where *library-name* represents the name of the shared library, and *entry-point* represents the entry point of the shared library for the stored procedure. If the name of the shared library and the entry point are the same, you can issue CALL <*entry-point*>.

NOT FENCED and regular FENCED stored procedures complicate your debugging efforts by giving your debugger access to additional address space. Local FENCED stored procedures run in the application's address space and allow your debugger to access both the application code and the stored procedure code. To enable local FENCED stored procedures for debugging, perform the following steps:

1. Register the stored procedure as a FENCED stored procedure.
2. Set the `DB2_STPROC_ALLOW_LOCAL_FENCED` registry variable to `true`. For information on registry variables, refer to the *Administration Guide: Implementation*.
3. Run the client application on the same machine as the DB2 server.

Note: While debugging a local FENCED stored procedure, be careful not to introduce statements that violate the restrictions listed in "Restrictions" on page 215. DB2 treats calls to local FENCED stored procedures as calls to a subroutine of the client application. Therefore, local FENCED stored procedures can contain statements that violate restrictions on normal stored procedures, like performing CONNECT statements in the procedure body.

When you write a NOT FENCED stored procedure, keep in mind 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.

Note: You should not use static data in stored procedures, because DB2 cannot guarantee that the static data in a stored procedure is or is not reinitialized on subsequent invocations.

NOT FENCED stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. See "The WCHARTYPE Precompiler Option in C and C++" on page 623 for more information.

DB2 does not support the use of any of the following features in NOT FENCED stored procedures:

- 16-bit
- Multi-threading
- Nested calls: calling or being called by another stored procedure
- Result sets: returning result sets to a client application or caller
- REXX

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

Returning Result Sets from Stored Procedures

You can code stored procedures to return one or more result sets to DB2 CLI, ODBC, JDBC, or SQLJ client applications. Aspects of this support include:

- Only DB2 CLI, ODBC, JDBC, and SQLJ clients can accept result sets.
- DB2 clients that use embedded SQL can accept multiple result sets if the stored procedure resides on a server that is accessible from a DataJoiner Version 2 server. Stored procedures on host and AS/400 platforms can return multiple result sets to DB2 Connect clients. Stored procedures on DB2 Universal Database servers can return multiple result sets to host and AS/400 clients. Consult the product documentation for DataJoiner or the host or AS/400 platform for more information.
- 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` for DB2 CLI, `getMoreResults` for JDBC, `getNextResultSet` for SQLJ) 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
- Only unread or unfetched rows are passed back in the result set.
- Stored procedures which return result sets must be run in FENCED mode.
- A COMMIT or ROLLBACK will close all cursors except WITH HOLD cursors.
- The RESULT_SETS column in the DB2CLI.PROCEDURES table indicates whether or not a stored procedure returns result sets. When you declare the stored procedure with the CREATE PROCEDURE statement, the DYNAMIC RESULT SETS clause sets this value to indicate the number of result sets returned by the stored procedure.

For additional details on handling result sets:

- in DB2 CLI, refer to the *CLI Guide and Reference*.
- in Java, refer to the DB2 Java Enablement web page at `http://www.ibm.com/software/data/db2/java/` for links to the JDBC and SQLJ specifications.

Example: Returning a Result Set from a Stored Procedure

This sample stored procedure shows how to return a result set to the client application in the following supported languages:

C	spserver.sqc
Java	Spserver.java

This sample stored procedure accepts one IN parameter and returns one OUT parameter and one result set. The stored procedure uses the IN parameter to create a result set containing the values of the NAME, JOB, and SALARY columns for the STAFF table for rows where SALARY is greater than the IN parameter.

1. Register the stored procedure using the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement. For example, to register the stored procedure written in embedded SQL for C, issue the following statement:

```
CREATE PROCEDURE RESULT_SET_CLIENT
   (IN salValue DOUBLE, OUT sqlCode INTEGER)
   DYNAMIC RESULT SETS 1
   LANGUAGE C
   PARAMETER STYLE GENERAL
   NO DBINFO
   FENCED
```

```
READS SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'spserver!one_result_set_to_client'
```

2. For embedded SQL in C stored procedures, use the DECLARE CURSOR and OPEN CURSOR statements to create an open cursor. For CLI stored procedures, use the `SQLPrepare` and `SQLBindParameter` APIs to create a result set. For Java stored procedures written with JDBC, use the `prepareStatement` and `executeQuery` methods to create a result set.
3. Close the connection to the database without closing the cursor or result set. This step does not apply to embedded SQL in C stored procedures.
4. Java stored procedures: for each result set that a PARAMETER STYLE JAVA stored procedure returns, you must include a corresponding *ResultSet[]* argument in the stored procedure method signature.

C Example: SPSERVER.SQC (one_result_set_to_client)

```
SQL_API_RC SQL_API_FN one_result_set_to_client
  (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  l_insalary = *insalary;
  *out_sqlerror = 0;

  EXEC SQL DECLARE c3 CURSOR FOR      2
    SELECT name, job, CAST(salary AS INTEGER)
    FROM staff
    WHERE salary > :l_insalary
    ORDER BY salary;

  EXEC SQL OPEN c3;                   2
  /* Leave cursor open to return result set */

  return (0);    3

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end one_result_set_to_client function */
```

Java Example: Spserver.java (resultSetToClient)

```
public static void resultSetToClient
     (double inSalaryThreshold, // double input
      int[] errorCode,            // SQLCODE output
      ResultSet[] rs)             // ResultSet output  4
      throws SQLException
{
   errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

   try {
      // Get caller's connection to the database
      Connection con =
        DriverManager.getConnection("jdbc:default:connection");

      // get salary result set using a parameter marker
      String query = "SELECT name, job, CAST(salary AS DOUBLE) " +
              "FROM staff " +
              "WHERE salary > ? " +
              "ORDER BY salary";

      // prepare the SQL statement
      PreparedStatement stmt = con.prepareStatement(query);

      // set the value of the parameter marker (?)
      stmt.setDouble(1, inSalaryThreshold);

      // get the result set that will be returned to the client
      rs[0] = stmt.executeQuery();  2

      // to return a result set to the client, do not close ResultSet
      con.close();  3
   }

   catch (SQLException sqle)
   {
      errorCode[0] = sqle.getErrorCode();
   }
}
```

Example: Accepting a Result Set from a Stored Procedure: This sample client application shows how to accept a result set from a stored procedure in the following supported languages:

C (using CLI) spclient.c

Java Spclient.java

This sample client application calls the `RESULT_SET_CLIENT` stored procedure and accepts one result set. The client application then displays the contents of the result set.

1. Call the stored procedure with arguments that correspond to the parameters you declared in the CREATE PROCEDURE statement.
2. JDBC applications use the `getNextResultSet` method to accept the first result set from the stored procedure.
3. Fetch the rows from the result set. The sample CLI client uses a `while` loop to fetch and display all rows from the result set. The sample JDBC client calls a class method called `fetchAll` that fetches and displays all rows from a result set.

CLI Example: SPCLIENT.C (one_result_set_to_client):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include <sqlca.h>
#include "utilcli.h"          /* Header file for CLI sample code */

     SQLCHAR      stmt[50];
     SQLINTEGER   out_sqlcode;
     char         out_buffer[33];
     SQLINTEGER   indicator;
     struct sqlca sqlca;
     SQLRETURN    rc,rc1  ;
     char         procname[254];
     SQLHANDLE    henv;  /* environment handle */
     SQLHANDLE    hdbc;  /* connection handle */
     SQLHANDLE    hstmt1; /* statement handle */
     SQLHANDLE    hstmt2; /* statement handle */
     SQLRETURN    sqlrc = SQL_SUCCESS;
     double       out_median;

     int oneresultset1(SQLHANDLE);

int main(int argc, char *argv[])
 {
     SQLHANDLE    hstmt; /* statement handle */
     SQLHANDLE    hstmt_oneresult; /* statement handle */

     char         dbAlias[SQL_MAX_DSN_LENGTH + 1] ;
     char         user[MAX_UID_LENGTH + 1] ;
     char         pswd[MAX_PWD_LENGTH + 1] ;

     /* Declare  variables for passing data to INOUT_PARAM */
     double inout_median;

    /* checks the command line arguments */
     rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );
     if ( rc != 0 )  return( 1 ) ;

    /* allocate an environment handle */
    printf("\n    Allocate an environment handle.\n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( sqlrc != SQL_SUCCESS )
    {   printf( "\n--ERROR while allocating the environment handle.\n" ) ;
        printf( "  sqlrc             = %d\n", sqlrc);
        printf( "  line              = %d\n", __LINE__);
        printf( "  file              = %s\n", __FILE__);
        return( 1 ) ;
    }

    /* allocate a database connection handle */
    printf("    Allocate a database connection handle.\n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;
```

```
HANDLE_CHECK( SQL_HANDLE_ENV, henv, sqlrc, &henv, &hdbc ) ;

/* connect to the database */
printf( "    Connecting to the database %s ...\n", dbAlias ) ;
sqlrc = SQLConnect( hdbc,
                (SQLCHAR *)dbAlias, SQL_NTS,
                (SQLCHAR *)user, SQL_NTS,
                (SQLCHAR *)pswd, SQL_NTS
              ) ;
HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
printf( "    Connected to the database %s.\n", dbAlias ) ;


/* set AUTOCOMMIT off */
sqlrc = SQLSetConnectAttr( hdbc,
                           SQL_ATTR_AUTOCOMMIT,
                           SQL_AUTOCOMMIT_OFF, SQL_NTS) ;
HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

/* allocate one or more statement handles */
printf("    Allocate a statement handle.\n");
sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt_oneresult ) ;
HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

/*******************************************************\
* Call oneresultsettocaller stored procedure            *
\*******************************************************/
 rc = oneresultset1(hstmt_oneresult);
 rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt_oneresult ) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

 /* ROLLBACK, free resources, and exit */

  rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  printf("\nStored procedure rolled back.\n\n");

 /* Disconnect from Remote Database */

  rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  printf( "\n>Disconnecting .....\n" ) ;
  rc = SQLDisconnect( hdbc ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
  if ( rc != SQL_SUCCESS ) return( SQL_ERROR )  ;
```

```
    return( SQL_SUCCESS ) ;
}

    int oneresultset1(hstmt)
    SQLHANDLE    hstmt; /* statement handle */
  {
  /*****************************************************\
  * Call one_result_set_to_client stored procedure      *
  \*****************************************************/

   double          insalary = 20000;
   SQLINTEGER      salary_int;
   SQLSMALLINT     num_cols;
   char            name[40];
   char            job[10];

   strcpy(procname, "RESULT_SET_CALLER");                                    1

   printf("\nCALL stored procedure:  %s\n", procname);

   strcpy((char*)stmt,"CALL RESULT_SET_CALLER ( ?,? )");
   rc = SQLPrepare(hstmt, stmt, SQL_NTS);
   STMT_HANDLE_CHECK( hstmt, rc);

   /* Bind the parameter to application variables () */
   rc = SQLBindParameter(hstmt, 1,
                         SQL_PARAM_INPUT, SQL_C_DOUBLE,
                         SQL_DOUBLE,0,
                         0, &insalary,
                         0, NULL);
   rc = SQLBindParameter(hstmt, 2,
                         SQL_PARAM_OUTPUT, SQL_C_LONG,
                         SQL_INTEGER,0,
                         0, &out_sqlcode,
                         0, NULL);

   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLExecute(hstmt);
   rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
   STMT_HANDLE_CHECK( hstmt, rc);

   rc = SQLNumResultCols( hstmt, &num_cols ) ;
   STMT_HANDLE_CHECK( hstmt, rc);
   printf("Result set returned %d columns\n", num_cols);

   /* bind columns to variables */
   rc = SQLBindCol( hstmt, 1, SQL_C_CHAR, name, 40, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLBindCol( hstmt, 2, SQL_C_CHAR, job, 10, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLBindCol( hstmt, 3, SQL_C_LONG, &salary_int, 0, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);

   /* fetch result set returned from stored procedure */
   rc = SQLFetch( hstmt );                                                   2
```

```
    rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);

    STMT_HANDLE_CHECK( hstmt, rc);

    printf("\n--------Name---------,  --JOB--, ---Salary--  \n");
    while (rc  == SQL_SUCCESS && rc != SQL_NO_DATA_FOUND )                    3
    {
    printf("%20s,%10s,    %d\n",name,job,salary_int);

    rc = SQLFetch( hstmt );
    }

    STMT_HANDLE_CHECK( hstmt, rc);

   /* Check that the stored procedure executed successfully */
    if (rc == SQL_SUCCESS) {
     printf("Stored procedure returned successfully.\n");
     }
     else {
     printf("Stored procedure returned SQLCODE %d\n", out_sqlcode);
     }
    rc = SQLCloseCursor(hstmt);

    return(rc);
}
```

Java Example: Spclient.java (resultSetToClient):

```
// prepare the CALL statement for RESULT_SET_CLIENT
procName = "RESULT_SET_CLIENT";
sql = "CALL " + procName + "(?, ?)"; 1
callStmt = con.prepareCall(sql);

// set input parameter to median value passed back by OUT_PARAM
callStmt.setDouble (1, outMedian);

// register the output parameter
callStmt.registerOutParameter (2, Types.INTEGER);

// call the stored procedure
System.out.println ("\nCall stored procedure named " + procName);
callStmt.execute();

// retrieve output parameter
outErrorCode = callStmt.getInt(2);

if (outErrorCode == 0) {
   System.out.println(procName + " completed successfully");
   ResultSet rs = callStmt.getResultSet(); 2
   while (rs.next()) {
      fetchAll(rs); 3
   }

   // close ResultSet
   rs.close();
}
else  { // stored procedure failed
   System.out.println(procName + " failed with SQLCODE "
     + outErrorCode);
}
```

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 stored in the proper location in accordance with the instructions in the *Application Building Guide*.

  For example, in an OS/2 environment, the dynamic link library for a FENCED stored procedure is located in the *instance_name*\function directory on the database server.
- The application, except if it is written in DB2 CLI and JDBC, is bound to the database.
- The stored procedure accurately returns any SQLCA error information to the client application.
- Stored procedure function names are *case-sensitive*, and must match exactly on client and server.
- If you register the stored procedure with a CREATE PROCEDURE statement, stored procedure function names must not match their library name.

  For example, the database manager will execute the stored procedure myfunc contained in the Windows 32-bit operating system library myfunc.dll as a DB2DARI function, disregarding the values specified in its associated CREATE PROCEDURE statement.

Note: For more information on debugging Java stored procedures, see "Debugging Stored Procedures in Java" on page 665.

You can use the debugger supplied with your compiler to debug a local FENCED stored procedure as you would any other application. Consult your compiler documentation for information on using the supplied debugger.

For example, to use the debugger supplied with Visual Studio™ on Windows NT, perform the following steps:

Step 1. Set the DB2_STPROC_ALLOW_LOCAL_FENCED registry variable to true.

Step 2. Compile the source file for the stored procedure DLL with the -Zi and -Od flags, and then link the DLL using the -DEBUG option.

Step 3. Copy the resulting DLL to the *instance_name* \function directory of the server.

Step 4. Invoke the client application on the server with the Visual Studio debugger. For the client application outcli.exe, enter the following command:

```
msdev spclient.exe
```

Step 5. When the Visual Studio debugger window opens, select **Project —> Settings**.

Step 6. Click the **Debug** tab.

Step 7. Click the **Category** arrow and select the **Additional DLLs**.

Step 8. Click the **New** button to create a new module.

Step 9. Click the **Browse** button to open the **Browse** window.

Step 10. Select the module `spserver.dll` and click **OK** to close the **Settings** window.

Step 11. Open the source file for the stored procedure and set a breakpoint.

Step 12. Click the Go button. The Visual Studio debugger stops when the stored procedure is invoked.

Step 13. At this point, you can debug the stored procedure using the Visual Studio debugger.

Refer to the Visual Studio product documentation for further information on using the Visual Studio debugger.

Chapter 8. Writing SQL Procedures

An *SQL procedure* is a stored procedure in which the procedural logic is contained in a CREATE PROCEDURE statement. The part of the CREATE PROCEDURE statement that contains the code is called the *procedure body*.

To create an SQL procedure, simply issue the CREATE PROCEDURE statement like any other DDL statement. You can also use the IBM DB2 Stored Procedure Builder to help you define the stored procedure to DB2, specify the source statements for the SQL procedure, and prepare the procedure for execution. For more information on the IBM DB2 Stored Procedure Builder, see "Chapter 9. IBM DB2 Stored Procedure Builder" on page 269.

This chapter discusses how to write a CREATE PROCEDURE statement that includes a procedure body. For more information on the syntax of the CREATE PROCEDURE statement and the procedure body, refer to the *SQL Reference*. For more information on using the IBM DB2 Stored Procedure Builder to create SQL procedures, see "Chapter 9. IBM DB2 Stored Procedure Builder" on page 269.

Comparison of SQL Procedures and External Procedures

Like external stored procedure definitions, SQL procedure definitions provide the following information:

- The procedure name.
- Parameter attributes.
- The language in which the procedure is written. For an SQL procedure, the language is SQL.

- Other information about the procedure, such as the specific name of the procedure and the number of result sets returned by the procedure.

Unlike a CREATE PROCEDURE statement for an external stored procedure, the CREATE PROCEDURE statement for an SQL procedure does not specify the EXTERNAL clause. Instead, an SQL procedure has a procedure body, which contains the source statements for the stored procedure.

The following example shows a CREATE PROCEDURE statement for a simple stored procedure. The procedure name, the list of parameters that are passed to or from the procedure, and the LANGUAGE parameter are common to all stored procedures. However, the LANGUAGE value of SQL and the BEGIN...END block, which forms the procedure body, are particular to an SQL procedure.

```
CREATE PROCEDURE UPDATE_SALARY_1          1
(IN EMPLOYEE_NUMBER CHAR(6),              2
IN RATE INTEGER)                          2
LANGUAGE SQL                              3
BEGIN
   UPDATE EMPLOYEE                        4
   SET SALARY = SALARY * (1.0 * RATE / 100.0 )
   WHERE EMPNO = EMPLOYEE_NUMBER;
END
```

Notes for the previous example:

1. The stored procedure name is UPDATE_SALARY_1.
2. The two parameters have data types of CHAR(6) and INTEGER. Both are input parameters.
3. LANGUAGE SQL indicates that this is an SQL procedure, so a procedure body follows the other parameters.
4. The procedure body consists of a single SQL UPDATE statement, which updates rows in the employee table.

Within the SQL procedure body, you cannot use OUT parameters as a value in any expression. You can only assign values to OUT parameters using the assignment statement, or as the target variable in the INTO clause of SELECT, VALUES and FETCH statements. You cannot use IN parameters as the target of assignment or INTO clauses.

Valid SQL Procedure Body Statements

A procedure body consists of a single SQL procedure statement. The types of statements that you can use in a procedure body include:

Assignment statement
Assigns a value to an output parameter or to an SQL variable, which is a

variable that is defined and used only within a procedure body. You cannot assign values to IN parameters.

CASE statement
Selects an execution path based on the evaluation of one or more conditions. This statement is similar to the CASE expression described in the *SQL Reference*.

FOR statement
Executes a statement or group of statements for each row of a table.

GET DIAGNOSTICS statement
The GET DIAGNOSTICS statement returns information about the previous SQL statement.

GOTO statement
Transfers program control to a user-defined label within an SQL routine.

IF statement
Selects an execution path based on the evaluation of a condition.

ITERATE statement
Passes the flow of control to a labelled block or loop.

LEAVE statement
Transfers program control out of a loop or block of code.

LOOP statement
Executes a statement or group of statements multiple times.

REPEAT statement
Executes a statement or group of statements until a search condition is true.

RESIGNAL statement
The RESIGNAL statement is used within a condition handler to resignal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

RETURN statement
Returns control from the SQL procedure to the caller. You can also return an integer value to the caller.

SIGNAL statement
The SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

SQL statement
The SQL procedure body can contain any SQL statement listed in "Appendix A. Supported SQL Statements" on page 737.

WHILE statement
Repeats the execution of a statement or group of statements while a specified condition is true.

Compound statement
Can contain one or more of any of the other types of statements in this list, as well as SQL variable declarations, condition handlers, or cursor declarations.

For a complete list of the SQL statements allowed within an SQL procedure body, see "Appendix A. Supported SQL Statements" on page 737. For detailed descriptions and syntax of each of these statements, refer to the *SQL Reference*.

Issuing CREATE PROCEDURE Statements

To issue a CREATE PROCEDURE statement as a DB2 Command Line Processor (DB2 CLP) script, you must use an alternate terminating character for SQL statements in the script. The semicolon (';') character, the default for DB2 CLP scripts, terminates SQL statements within the SQL procedure body.

To use an alternate terminating character in DB2 CLP scripts, select a character that is not used in standard SQL statements. In the following example, the at sign ('@') is used as the terminating character for a DB2 CLP script named script.db2:

```
CREATE PROCEDURE UPDATE_SALARY_IF
(IN employee_number CHAR(6), IN rating SMALLINT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
     SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

  IF (rating = 1)
    THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
  ELSEIF (rating = 2)
    THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
  ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
  END IF;
END
@
```

To process the DB2 CLP script from the command line, use the following syntax:

```
db2 -td term-char -vf script-name
```

where *term-char* represents the terminating character, and *script-name* represents the name of the DB2 CLP script to process. To process the preceding script, for example, issue the following command from the CLP:

```
db2 -td@ -vf script.db2
```

Handling Conditions in SQL Procedures

Condition handlers determine the behavior of your SQL procedure when a condition occurs. You can declare one or more condition handlers in your SQL procedure for general DB2 conditions, defined conditions for specific SQLSTATE values, or specific SQLSTATE values. For more information on general conditions and on defining your own conditions, see "Declaring Condition Handlers".

If a statement in your SQL procedure issues an SQLWARNING or NOT FOUND condition, and you have declared a handler for the respective condition, DB2 passes control to the corresponding handler. If you have not declared a handler for that particular condition, DB2 sets the variables SQLSTATE and SQLCODE with the corresponding values for the condition and passes control to the next statement in the procedure body.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 passes control to that handler. If DB2 successfully executes the handler, the values of SQLSTATE and SQLCODE return '00000' and 0 respectively.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you have not declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 terminates the SQL procedure and returns to the client.

Declaring Condition Handlers

The general form of a handler declaration is:

```
DECLARE handler-type  HANDLER FOR condition SQL-procedure-statement
```

When DB2 raises a condition that matches *condition*, DB2 passes control to the condition handler. The condition handler performs the action indicated by *handler-type*, and then executes *SQL-procedure-statement*.

handler-type

CONTINUE
Specifies that after *SQL-procedure-statement* completes, execution continues with the statement after the statement that caused the error.

EXIT Specifies that after *SQL-procedure-statement* completes, execution continues at the end of the compound statement that contains the handler.

UNDO
Specifies that before *SQL-procedure-statement* executes, DB2 rolls back any SQL operations that have occurred in the compound statement that contains the handler. After *SQL-procedure-statement* completes, execution continues at the end of the compound statement that contains the handler.

Note: You can only declare UNDO handlers in ATOMIC compound statements.

condition
DB2 provides three general conditions:

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 or an SQLSTATE of '02000'.

SQLEXCEPTION
Identifies any condition that results in a negative SQLCODE.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is 'W'), or that results in a positive SQL return code other than +100.

You can also use the DECLARE statement to define your own condition for a specific SQLSTATE. For more information on defining your own condition, refer to the *SQL Reference*.

SQL-procedure-statement
You can use a single SQL procedure statement to define the behavior of the condition handler. DB2 accepts a compound statement delimited by a BEGIN...END block as a single SQL procedure statement. If you use a compound statement to define the behavior of a condition handler, and you want the handler to retain the value of either the SQLSTATE or SQLCODE variables, you must assign the value of the variable to a local variable or parameter in the first statement of the compound block. If the first statement of a compound block does not assign the value of SQLSTATE or SQLCODE to a local variable or parameter, SQLSTATE and SQLCODE cannot retain the value that caused DB2 to invoke the condition handler.

Note: You cannot define another condition handler within the condition handler.

The following examples demonstrate simple condition handlers:

Example: CONTINUE handler: This handler assigns the value of 1 to the local variable *at_end* when DB2 raises a NOT FOUND condition. DB2 then passes control to the statement following the one that raised the NOT FOUND condition.

```
DECLARE not_found CONDITION FOR SQLSTATE '02000';
DECLARE CONTINUE HANDLER FOR not_found SET at_end=1;
```

Example: EXIT handler: The procedure declares NO_TABLE as the condition name for SQLSTATE 42704 (*name* is an undefined name). The condition handler for NO_TABLE places the string `Table does not exist` into output parameter OUT_BUFFER. The handler then causes the SQL procedure to exit the compound statement in which the handler is declared.

```
DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
DECLARE EXIT HANDLER FOR NO_TABLE
   BEGIN
       SET OUT_BUFFER='Table does not exist';
   END
```

Example: UNDO handler: The procedure declares an UNDO condition handler for SQLSTATE 42704 without first defining a name for the SQLSTATE. The handler causes the SQL procedure to roll back the current unit of work, place the string `Table does not exist` into output parameter OUT_BUFFER, and exit the compound statement in which the handler is declared.

```
DECLARE UNDO HANDLER FOR SQLSTATE '42704'
   BEGIN
       SET OUT_BUFFER='Table does not exist';
   END;
```

Note: You can only declare UNDO handlers in ATOMIC compound statements.

SIGNAL and RESIGNAL Statements

You can use the SIGNAL and RESIGNAL statements to explicitly raise a specific SQLSTATE. Use the SET MESSAGE_TEXT clause of the SIGNAL and RESIGNAL statements to define the text that DB2 displays for a custom-defined SQLSTATE.

In the following example, the SQL procedure body declares a condition handler for the custom SQLSTATE `72822`. When the procedure executes the SIGNAL statement that raises SQLSTATE `72822`, DB2 invokes the condition handler. The condition handler tests the value of the SQL variable *var* with an IF statement. If *var* is `OK`, the handler redefines the SQLSTATE value as `72623` and assigns a string literal to the text associated with SQLSTATE 72623. If *var* is not `OK`, the handler redefines the SQLSTATE value as `72319` and assigns the value of *var* to the text associated with that SQLSTATE.

```
DECLARE EXIT CONDITION HANDLER FOR SQLSTATE '72822'
   BEGIN
       IF ( var = 'OK' )
          RESIGNAL '72623' SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
       ELSE
          RESIGNAL '72319' SET MESSAGE_TEXT = var;
   END;

SIGNAL SQLSTATE '72822';
```

For more information on the SIGNAL and RESIGNAL statements, refer to the *SQL Reference*.

SQLCODE and SQLSTATE Variables in SQL Procedures

To help debug your SQL procedures, you might find it useful to insert the value of the SQLCODE and SQLSTATE into a table at various points in the SQL procedure, or to return the SQLCODE and SQLSTATE values in a diagnostic string as an OUT parameter. To use the SQLCODE and SQLSTATE values, you must declare the following SQL variables in the SQL procedure body:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

You can also use CONTINUE condition handlers to assign the value of the SQLSTATE and SQLCODE variables to local variables in your SQL procedure body. You can then use these local variables to control your procedural logic, or pass the value back as an output parameter. In the following example, the SQL procedure returns control to the statement following each SQL statement with the SQLCODE set in a local variable called RETCODE.

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE retcode INTEGER DEFAULT 0;

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET retcode = SQLCODE;
```

Note: When you access the SQLCODE or SQLSTATE variables in an SQL procedure, DB2 sets the value of SQLCODE to 0 and SQLSTATE to '00000' for the subsequent statement.

Using Dynamic SQL in SQL Procedures

SQL procedures, like external stored procedures, can issue dynamic SQL statements. If your dynamic SQL statement does not include parameter markers and you plan to execute it only once, use the EXECUTE IMMEDIATE statement.

If your dynamic SQL statement contains parameter markers, you must use the PREPARE and EXECUTE statements. If you you plan to execute a dynamic SQL statement multiple times, it might be more efficient to issue a single PREPARE statement and to issue the EXECUTE statement multiple times rather than issuing the EXECUTE IMMEDIATE statement each time. To use the PREPARE and EXECUTE statements to issue dynamic SQL in your SQL procedure, you must include the following statements in the SQL procedure body:

Step 1. Declare a variable of type VARCHAR that is large enough to hold your dynamic SQL statement using a DECLARE statement.

Step 2. Assign a statement string to the variable using a SET statement. You cannot include variables directly in the statement string. Instead, you must use the question mark ('?') symbol as a parameter marker for any variables used in the statement.

Step 3. Create a prepared statement from the statement string using a PREPARE statement.

Step 4. Execute the prepared statement using an EXECUTE statement. If the statement string includes a parameter marker, use a USING clause to replace it with the value of a variable.

Note: Statement names defined in PREPARE statements for SQL procedures are treated as scoped variables. Once the SQL procedure exits the scope in which you define the statement name, DB2 can no longer access the statement name. Inside any compound statement, you cannot issue two PREPARE statements that use the same statement name.

Example: Dynamic SQL statements: The following example shows an SQL procedure that includes dynamic SQL statements.

The procedure receives a department number (*deptNumber*) as an input parameter. In the procedure, three statement strings are built, prepared, and executed. The first statement string executes a DROP statement to ensure that the table to be created does not already exist. This table is named DEPT_*deptno*_T, where *deptno* is the value of input parameter *deptNumber*. A CONTINUE HANDLER ensures that the SQL procedure will continue if it detects SQLSTATE 42704 (“undefined object name”), which DB2 returns from the DROP statement if the table does not exist. The second statement string issuees a CREATE statement to create DEPT_*deptno*_T. The third statement string inserts rows for employees in department *deptno* into DEPT_*deptno*_T. The third statement string contains a parameter marker that represents *deptNumber*. When the prepared statement is executed, parameter *deptNumber* is substituted for the parameter marker.

```
CREATE PROCEDURE create_dept_table
(IN deptNumber VARCHAR(3), OUT table_name VARCHAR(30))
LANGUAGE SQL
```

```
BEGIN
  DECLARE stmt VARCHAR(1000);

  -- continue if sqlstate 42704 ('undefined object name')
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42704'
    SET stmt = '';
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET table_name = 'PROCEDURE_FAILED';

  SET table_name = 'DEPT_'||deptNumber||'_T';
  SET stmt = 'DROP TABLE '||table_name;
  PREPARE s1 FROM stmt;
  EXECUTE s1;
  SET stmt = 'CREATE TABLE '||table_name||
   '( empno CHAR(6) NOT NULL, '||
   'firstnme VARCHAR(12) NOT NULL, '||
   'midinit CHAR(1) NOT NULL, '||
   'lastname VARCHAR(15) NOT NULL, '||
   'salary DECIMAL(9,2))';
  PREPARE s2 FROM STMT;
  EXECUTE s2;
  SET stmt = 'INSERT INTO '||table_name || ' ' ||
   'SELECT empno, firstnme, midinit, lastname, salary '||
   'FROM employee '||
   'WHERE workdept = ?';
 PREPARE s3 FROM stmt;
 EXECUTE s3 USING deptNumber;
END
```

Nested SQL Procedures

Your SQL procedures can contain CALL statements to call other SQL procedures. This feature, called *nested* stored procedures, enables you to reuse existing SQL procedures and design more complex applications.

Passing Parameters Between Nested SQL Procedures

To call a *target* SQL procedure from within a *caller* SQL procedure, simply include a CALL statement with the appropriate number and types of parameters in your caller. If the target returns OUT parameters, the caller can use the returned values in its own statements.

For example, you can create an SQL procedure that calls a target SQL procedure with the name "SALES_TARGET" and that accepts a single OUT parameter of type INTEGER with the following SQL:

```
CREATE PROCEDURE NEST_SALES(OUT budget DECIMAL(11,2))
LANGUAGE SQL
BEGIN
   DECLARE total INTEGER DEFAULT 0;
   SET total = 6;
   CALL SALES_TARGET(total);
   SET budget = total * 10000;
END
```

Returning Result Sets From Nested SQL Procedures

If a target SQL procedure returns result sets, either the caller or the client application receives the result sets, depending on the DECLARE CURSOR statements issued by the target SQL procedure. For each DECLARE CURSOR statement in the target that includes the WITH RETURN TO CLIENT clause, the caller does not receive the result set. For WITH RETURN TO CLIENT cursors, the result set is returned directly to the client application.

For more information on returning result sets from nested SQL procedures, see "Returning Result Sets to Caller or Client" on page 258.

Restrictions on Nested SQL Procedures

Keep the following restrictions in mind when designing your application architecture:

LANGUAGE
SQL procedures can only call stored procedures written in SQL or C. You cannot call other host language stored procedures from within an SQL procedure.

16 levels of nesting
You may only include a maximum of 16 levels of nested calls to SQL procedures. A scenario where SQL procedure A calls SQL procedure B, and SQL procedure B calls SQL procedure C, is an example of three levels of nested calls.

Recursion
You can create an SQL procedure that calls itself recursively. Recursive SQL procedures must comply with the previously described restriction on the maximum levels of nesting.

Security
An SQL procedure cannot call a target SQL procedure that is cataloged with a higher SQL data access level. For example, an SQL procedure created with the CONTAINS SQL clause can call SQL procedures created with either the CONTAINS SQL clause or the NO SQL clause, and cannot call SQL procedures created with either the READS SQL DATA clause or the MODIFIES SQL DATA clause.

An SQL procedure created with the NO SQL clause cannot issue a CALL statement.

Returning Result Sets From SQL Procedures

Returning result sets from SQL procedures is similar to returning result sets from external stored procedures. Client applications must use the CLI, JDBC, or SQLJ application programming interfaces to accept result sets from an SQL procedure. SQL procedures that call other SQL procedures also can accept

result sets from those procedures. To return a result set from an SQL procedure, write your SQL procedure as follows:

1. Declare the number of result sets that the SQL procedure returns using the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement.
2. Declare a cursor using the DECLARE CURSOR statement.
3. Open the cursor using the OPEN CURSOR statement.
4. Exit from the SQL procedure without closing the cursor.

For example, you can write an SQL procedure that returns a single result set, based on the value of the INOUT parameter *threshold*, as follows:

```
CREATE PROCEDURE RESULT_SET (INOUT threshold SMALLINT)
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN
   DECLARE cur1 CURSOR WITH RETURN TO CALLER FOR
       SELECT name, job, years
       FROM staff
       WHERE years < threshold;
   OPEN cur1;
END
```

Returning Result Sets to Caller or Client

If your application returns result sets from nested SQL procedures, you must use the WITH RETURN clause of the DECLARE CURSOR statement to ensure that DB2 returns the result sets to the appropriate location. If a target SQL procedure returns result sets to a calling SQL procedure, the caller must use the ALLOCATE CURSOR and ASSOCIATE RESULT SET LOCATOR statements to access and use the result set.

Returning Result Sets to the Client

To always return a result set from an SQL procedure to a client application, use the WITH RETURN TO CLIENT clause in the DECLARE CURSOR statement associated with the result set. In the following example, the SQL procedure "CLIENT_SET" uses the WITH RETURN TO CLIENT clause in the DECLARE CURSOR statement to return a result set to the client application, even if "CLIENT_SET" is the target of a nested SQL procedure CALL statement:

```
CREATE PROCEDURE CLIENT_SET()
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
   DECLARE clientcur CURSOR WITH RETURN TO CLIENT
       FOR SELECT name, dept, job
       FROM staff
       WHERE salary > 20000;
   OPEN clientcur;
END
```

Returning Result Sets to the Caller

To return a result set to the direct caller of an SQL procedure, whether the caller is a client application or another SQL procedure, use the WITH RETURN TO CALLER clause in the DECLARE CURSOR statement associated with the result set. In the following example, the SQL procedure "CALLER_SET" uses the WITH RETURN TO CALLER clause to return a result set to the caller of CALLER_SET:

```
CREATE PROCEDURE CALLER_SET()
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
   DECLARE clientcur CURSOR WITH RETURN TO CALLER
       FOR SELECT name, dept, job
       FROM staff
       WHERE salary > 15000;
   OPEN clientcur;
END
```

Receiving Result Sets as a Caller

When you expect your calling SQL procedure to receive a result set from a target SQL procedure, you must use the ALLOCATE CURSOR and ASSOCIATE RESULT SET LOCATOR statements to access and use the result set.

ASSOCIATE RESULT SET LOCATOR

After a CALL statement to a target SQL procedure that returns one or more result sets to the caller, your calling SQL procedure should issue this statement to assign result set locator variables for each of the returned result sets. For example, a calling SQL procedure that expects to receive three result sets from a target SQL procedure could contain the following SQL:

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;

CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
```

ALLOCATE CURSOR

Use the ALLOCATE CURSOR statement in a calling SQL procedure to open a result set returned from a target SQL procedure. To use the ALLOCATE CURSOR statement, the result set must already be associated with a result set locator through the ASSOCIATE RESULT SET LOCATORS statement. Once the SQL procedure issues an ALLOCATE CURSOR statement, you can fetch rows from the result set using the cursor name declared in the ALLOCATE CURSOR statement. To extend the previously described ASSOCIATE LOCATORS example, the SQL procedure could fetch rows from the first of the returned result sets using the following SQL:

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
ALLOCATE rsCur CURSOR FOR result1;
WHILE (at_end = 0) DO
   SET total1 = total1 + var1;
   SET total2 = total2 + var2;
   FETCH FROM rsCur INTO var1, var2;
END WHILE;
```

Debugging SQL Procedures

After writing your SQL procedure, you must issue the CREATE PROCEDURE statement as described in "Issuing CREATE PROCEDURE Statements" on page 250. In certain situations, DB2 may return an error in response to your CREATE PROCEDURE statement. To retrieve more information on the error returned by DB2, including an explanation and suggestions for correcting the error, issue the following command at the CLP:

```
db2 "? error-code"
```

where *error-code* represents the SQLCODE or SQLSTATE returned by the error. For example, if your CREATE PROCEDURE statement returns an error with SQLCODE "SQL0469N" ("Parameter mode is not valid"), issue the following command:

```
db2 "? SQL0469"
```

DB2 returns the following message:

```
Explanation:  One of the following errors occurred:

o   a parameter in an SQL procedure is declared as OUT and is
    used as input in the procedure body

o   a parameter in an SQL procedure is declared as IN and is
    modified in the procedure body.

User Response:  Change the attribute of the parameter to INOUT,
or change the use of the parameter within the procedure.
```

Once you display the message, try modifying your SQL procedure following the suggestions in the "User Response" section.

Displaying Error Messages for SQL Procedures

When you issue a CREATE PROCEDURE statement for an SQL procedure, DB2 may accept the syntax of the SQL procedure body but fail to create the SQL procedure at the precompile or compile stage. In these situations, DB2

normally creates a log file that contains the error messages. This log file, and other intermediate files, are described in "Debugging SQL Procedures Using Intermediate Files" on page 263.

To retrieve the error messages generated by DB2 and the C compiler for an SQL procedure, display the message log file in the following directory on your database server:

UNIX *$DB2PATH*/function/routine/sqlproc/*$DATABASE*/*$SCHEMA*/tmp

where *$DB2PATH* represents the location of the instance directory, *$DATABASE* represents the database name, and *$SCHEMA* represents the schema name used to create the SQL procedure.

Windows NT

%DB2PATH%\function\routine\sqlproc*%DB%**%SCHEMA%*\tmp

where *%DB2PATH%* represents the location of the instance directory, *%DB%* represents the database name, and *%SCHEMA%* represents the schema name used to create the SQL procedure.

You can also issue a CALL statement in an application to call the sample stored procedure `db2udp!get_error_messages` using the following syntax:

```
CALL db2udp!get_error_messages(schema-name, file-name, message-text)
```

where *schema-name* is an input parameter representing the schema of the SQL procedure, *file-name* is an input parameter representing the generated file name for the SQL procedure, and *message-text* is an output parameter that returns the message text in the message log file.

For example, you could use the following Java application to display the error messages for an SQL procedure:

```
public static String getErrorMessages(Connection  con,
  String procschema, String filename) throws Exception
{
    String filecontents = null;
    // prepare the CALL statement
    CallableStatement stmt = null;
    try
    {
        String sql = "Call db2udp!get_error_messages(?, ?, ?) ";
        stmt = con.prepareCall (sql);

        // set all parameters (input and output)
        stmt.registerOutParameter( 3, java.sql.Types.LONGVARCHAR  );
        stmt.setString( 1, procschema );
        stmt.setString( 2, filename );

        // call the stored procedure
        boolean isrs = stmt.execute();
        filecontents = stmt.getString(3);
```

```
            System.out.println("SQL Procedure - getErrorMessages "
              + filecontents);
            return filecontents;
        }
        catch (Exception e) { throw e; }
        finally
        {
            if (stmt != null) stmt.close();
        }
    }
```

You could use the following C application to display the error messages for an SQL procedure:

```
int getErrors(char inputSchema[9], char inputFilename[9],
                        char outputFilecontents[32000])
{
  EXEC SQL BEGIN DECLARE SECTION;
    char    procschema[100] = "";
    char    filename[100] = "";
    char    filecontents[32000] = "";
  EXEC SQL END DECLARE SECTION;

  strcpy (procschema, inputSchema);
  strcpy (filename, inputFilename);

  EXEC SQL CALL "db2udp!get_error_messages"
    (:procschema, :filename, :filecontents);
  if ( sqlca.sqlcode != 0 )
  {
    printf("Call failed. Code: %d\n", sqlca.sqlcode);
    return 1;
  }
  else
  {
    printf("\nSQL Procedure - getErrors:\n%s\n", filecontents);
  }
  strcpy (outputFilecontents, filecontents);
  return 0;
}
```

Note: Before you can display the error messages for an SQL procedure that DB2 failed to create, you must know both the procedure name and the generated file name of the SQL procedure. If the procedure schema name is not issued as part of the CREATE PROCEDURE statement, DB2 uses the value of the CURRENT SCHEMA special register. To display the value of the CURRENT SCHEMA special register, issue the following statement at the CLP:

```
VALUES CURRENT SCHEMA
```

Debugging SQL Procedures Using Intermediate Files

When you issue a CREATE PROCEDURE statement for an SQL procedure, and DB2 accepts the syntax of the SQL procedure body, DB2 uses a number of intermediate files to create the SQL procedure. After DB2 successfully creates an SQL procedure, it normally removes the intermediate files to conserve system resources. If DB2 accepts the CREATE PROCEDURE syntax, but fails to create an SQL procedure, it retains a log file that tracks the precompile, bind, and compile stages of the CREATE PROCEDURE process.

On UNIX systems, DB2 uses the following base directory to keep intermediate files: *instance*/function/routine/sqlproc/*dbAlias*/*schema*, where *instance* represents the path of the DB2 instance, *dbAlias* represents the database alias, and *schema* represents the schema with which the CREATE PROCEDURE statement was issued.

On OS/2 and Windows 32-bit operating systems, DB2 uses the following base directory to keep intermediate files: *instance*\function\routine\sqlproc*dbAlias**schema*, where *instance* represents the path of the DB2 instance, *dbAlias* represents the database alias, and *schema* represents the schema with which the CREATE PROCEDURE statement was issued.

If the SQL procedure was created successfully, but does not return the expected results from your CALL statements, you may want to examine the intermediate files. To prevent DB2 from removing the intermediate files, set the DB2_SQLROUTINE_KEEP_FILES DB2 registry variable to "yes" using the following command:

```
db2set DB2_SQLROUTINE_KEEP_FILES="yes"
```

Before DB2 can use the new value of the registry variable, you must restart the database.

Examples of SQL Procedures

This section contains examples of how to use each of the statements that can appear in an SQL procedure body. For these and other example SQL procedures, including client applications that call the SQL procedures, refer to the following directories:

UNIX operating systems
$HOME*/sqllib/samples/sqlproc, where *$HOME represents the location of your DB2 instance directory

Windows 32-bit operating systems
%DRIVE%\sqllib\samples\sqlproc, where *%DRIVE%* represents the drive on which you installed DB2

Example 1: CASE statement: The following SQL procedure demonstrates how to use a CASE statement. The procedure receives the ID number and rating of an employee as input parameters. The CASE statement modifies the salary and bonus for the employee, using a different UPDATE statement for each of the possible ratings.

```
CREATE PROCEDURE UPDATE_SALARY
(IN employee_number CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
    SIGNAL SQLSTATE '02444';

  CASE rating
    WHEN 1 THEN
      UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
    WHEN 2 THEN
      UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
    ELSE
      UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
   END CASE;
END
```

Example 2: Compound statement with nested IF and WHILE statements: The following example shows a compound statement that includes nested IF statements, a WHILE statement, and assignment statements. The example also shows how to declare SQL variables, cursors, and handlers for classes of error codes.

The procedure receives a department number as an input parameter. A WHILE statement in the procedure body fetches the salary and bonus for each employee in the department. An IF statement within the WHILE statement updates salaries for each employee depending on number of years of service and current salary. When all employee records in the department have been processed, the FETCH statement that retrieves employee records receives SQLSTATE 20000. A *not_found* condition handler makes the search condition for the WHILE statement false, so execution of the WHILE statement ends.

```
CREATE PROCEDURE BUMP_SALARY_IF (IN deptnumber SMALLINT)
LANGUAGE SQL
BEGIN
   DECLARE v_salary DOUBLE;
   DECLARE v_years SMALLINT;
   DECLARE v_id SMALLINT;
   DECLARE at_end INT DEFAULT 0;
```

```
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  -- CAST salary as DOUBLE because SQL procedures do not support DECIMAL
  DECLARE C1 CURSOR FOR
    SELECT id, CAST(salary AS DOUBLE), years
    FROM staff;
  DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;

  OPEN C1;
  FETCH C1 INTO v_id, v_salary, v_years;
  WHILE at_end = 0 DO
    IF (v_salary < 2000 * v_years)
      THEN UPDATE staff
        SET salary = 2150 * v_years
        WHERE id = v_id;
      ELSEIF (v_salary < 5000 * v_years)
        THEN IF (v_salary < 3000 * v_years)
          THEN UPDATE staff
            SET salary = 3000 * v_years
            WHERE id = v_id;
        ELSE UPDATE staff
          SET salary = v_salary * 1.10
          WHERE id = v_id;
        END IF;
      ELSE UPDATE staff
        SET job = 'PREZ'
        WHERE id = v_id;
    END IF;
    FETCH C1 INTO v_id, v_salary, v_years;
  END WHILE;
  CLOSE C1;
END
```

Example 3: Using Nested SQL Procedures with Global Temporary Tables and Result Sets:

The following example shows how to use the ASSOCIATE RESULT SET LOCATOR and ALLOCATE CURSOR statements to return a result set from the called SQL procedure, `temp_table_insert`, to the calling SQL procedure, `temp_table_create`. The example also shows how a called SQL procedure can use a global temporary table that is created by a calling SQL procedure.

In the example, a client application or another SQL procedure calls `temp_table_create`, which creates the global temporary table SESSION.TTT and in turn calls `temp_table_insert`.

To use the SESSION.TTT global temporary table, `temp_table_insert` contains a DECLARE GLOBAL TEMPORARY TABLE statement identical to the statement that `temp_table_create` issues to create SESSION.TTT. The difference is that `temp_table_insert` contains the DECLARE GLOBAL

TEMPORARY TABLE statement within an IF statement that is always false. The IF statement prevents DB2 from attempting to create the global temporary table for a second time, but enables the SQL procedure to use the global temporary table in subsequent statements.

To return a result set from a global temporary table that was created by a different SQL procedure, `temp_table_insert` must issue the DECLARE CURSOR statement within a new scope. `temp_table_insert` issues the DECLARE CURSOR and OPEN CURSOR statements within a compound SQL block, which satisfies the requirement for a new scope. The cursor is not closed before the SQL procedure exits, so DB2 passes the result set back to the caller, `temp_table_create`.

To accept the result set from the called SQL procedure, `temp_table_create` issues an ASSOCIATE RESULT SET LOCATOR statement that identifies `temp_table_insert` as the originator of the result set. `temp_table_create` then issues an ALLOCATE CURSOR statement for the result set locator to open the result set. If the ALLOCATE CURSOR statement succeeds, the SQL procedure can work with the result set as usual. In this example, `temp_table_create` fetches every row from the result set, adding the values of the columns to its output parameters.

Note: Before issuing a CREATE PROCEDURE statement for an SQL procedure that uses global temporary tables, you must create a user temporary tablespace. To create a user temporary tablespace, issue the following SQL statement:

```
CREATE USER TEMPORARY TABLESPACE ts1
   MANAGED BY SYSTEM USING ('ts1file');
```

where *ts1* represents the name of the user temporary tablespace, and *ts1file* represents the name of the container used by the tablespace.

```
CREATE PROCEDURE temp_table_create(IN parm1 INTEGER, IN parm2 INTEGER,
   OUT parm3 INTEGER, OUT parm4 INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE loc1 RESULT_SET_LOCATOR VARYING;
      DECLARE total3,total4 INTEGER DEFAULT 0;
      DECLARE rcolumn1, rcolumn2 INTEGER DEFAULT 0;
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND, SQLEXCEPTION, SQLWARNING
      BEGIN
        SET result_set_end = 1;
      END;
      --Create the temporary table that is used in both this SQL procedure
      --and in the SQL procedure called by this SQL procedure.
      DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
         NOT LOGGED;
      --Insert rows into the temporary table.
      --The result set includes these rows.
```

```
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+1, parm2+1);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+2, parm2+2);
      --Make a nested call to the 'temp_table_insert' SQL procedure.
      CALL temp_table_insert(parm1, parm2);
      --Issue the ASSOCIATE RESULT SET LOCATOR statement to
      --accept a single result set from 'temp_table_insert'.
      --If 'temp_table_insert' returns multiple result sets,
      --you must declare one locator variable (for example,
      --ASSOCIATE RESULT SET LOCATOR(loc1, loc2, loc3) for each result set.
      ASSOCIATE RESULT SET LOCATOR(loc1) WITH PROCEDURE temp_table_insert;
      --The ALLOCATE statement is similar to the OPEN statement.
      --It makes the result set available in this SQL procedure.
      ALLOCATE cursor1 CURSOR FOR RESULT SET loc1;
      --Insert rows into the temporary table.
      --The result set does not include these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+5, parm2+5);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+6, parm2+6);
      SET result_set_end = 0;
      --Fetch the columns from the first row of the result set.
      FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      WHILE (result_set_end = 0) DO
        SET total3 = total3 + rcolumn1;
        SET total4 = total4 + rcolumn2;
        --Fetch columns from the result set for the
        --next iteration of the WHILE loop.
        FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      END WHILE;
      CLOSE cursor1;
      SET parm3 = total3;
      SET parm4 = total4;
   END @

CREATE PROCEDURE temp_table_insert (IN parm1 INTEGER, IN parm2 INTEGER )
   LANGUAGE SQL
   BEGIN
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND BEGIN
        SET result_set_end = 1;
      END;
      --To use a temporary table that is created by a different stored
      --procedure, include a DECLARE GLOBAL TEMPORARY TABLE statement
      --inside a condition statement that always evaluates to false.
      IF (1 = 0) THEN
        DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
           NOT LOGGED;
      END IF;
      --Insert rows into the temporary table.
      --The result set includes these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+3, parm2+3);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+4, parm2+4);
      --To return a result set from the temporary table, issue
      --the DECLARE CURSOR statement inside a new scope, such as
      --a compound SQL statement (BEGIN...END block).
      --Issue the DECLARE CURSOR statement after the DECLARE
      --GLOBAL TEMPORARY TABLE statement.
```

```
   BEGIN
      --The WITH RETURN TO CALLER clause causes the SQL procedure
      --to return its result set to the calling procedure.
      DECLARE cur1 CURSOR WITH RETURN TO CALLER
         FOR SELECT * FROM session.ttt;
      --To return a result set, open a cursor without closing the cursor.
     OPEN cur1 ;
   END;
END
```

Chapter 9. IBM DB2 Stored Procedure Builder

What is Stored Procedure Builder?

Stored Procedure Builder is a graphical application that supports the rapid development of DB2 stored procedures. Using Stored Procedure Builder, you can perform the following tasks:

- Create new stored procedures
- Build stored procedures on local and remote DB2 servers
- Modify and rebuild existing stored procedures
- Test and debug the execution of installed stored procedures

To create an application which has a stored procedure, Stored Procedure Builder provides a single development environment that supports the entire DB2 Universal Database family, including the OS/2, OS/390, OS/400, AIX, HP-UX, Linux, Solaris Operating Environment, and Windows 32-bit operating systems.

Supported Platforms for Stored Procedure Builder:

The Stored Procedure Builder is an optional component of the DB2 Application Development Client on AIX, Solaris** Operating Environment**, and the Windows 32-bit operating systems.

You can use Stored Procedure Builder on your client to build and deploy Java stored procedures and SQL procedures on DB2 Universal Database servers for the following platforms:

Stored Procedure Language	Supported DB2 UDB Platforms
Java	OS/2, OS/390, AIX, HP-UX, Linux, Solaris Operating Environment, and Windows 32-bit operating systems
SQL	OS/2, OS/390, OS/400, AIX, HP-UX, Linux, Solaris Operating Environment, and Windows 32-bit operating systems

You can export SQL stored procedures and create Java stored procedures from existing Java class files. To provide a comfortable development environment,

the Stored Procedure Builder code editor enables you to use `vi` or `emacs` key bindings, in addition to the default key bindings.

Launching Stored Procedure Builder:

On Windows 32-bit operating systems, you can launch Stored Procedure Builder from the DB2 Universal Database program group, issuing the `db2spb` command from the command line, or from any of the following development applications:

- Microsoft Visual C++ 5.0 and 6.0
- Microsoft Visual Basic 5.0 and 6.0
- IBM VisualAge for Java

On AIX and Solaris Operating Environment clients, you can launch Stored Procedure Builder by issuing the `db2spb` command from the command line.

Stored Procedure Builder is implemented with Java and all database connections are managed with Java Database Connectivity (JDBC). Using a JDBC driver, you can connect to any local DB2 alias or any other database for which you can specify a host, port, and database name.

Note: To use Stored Procedure Builder, you must be connected to a DB2 database for development. For more information about using Stored Procedure Builder, refer to the IBM DB2 Stored Procedure Builder online help.

Advantages of Using Stored Procedure Builder

Stored Procedure Builder provides an easy-to-use development environment for creating, installing, and testing stored procedures, helping you to focus on creating your stored procedure logic rather than the details of registering, building, and installing stored procedures on a DB2 server. Stored Procedure Builder helps you develop cross-platform applications by enabling you to build a stored procedure on server platforms that differ from the platform on which you develop the stored procedure.

Creating New Stored Procedures

Using Stored Procedure Builder greatly simplifies the process of creating and installing stored procedures on a DB2 database server. The stored procedure wizards and the SQL Assistant facilitate the development of stored procedures.

In Stored Procedure Builder you can create highly portable stored procedures written in Java or SQL. Using the stored procedure wizards, you create your

basic SQL structure and then use the source code editor to modify the stored procedure to contain sophisticated stored procedure logic.

When creating a stored procedure, you can choose to return a single result set, multiple result sets, or output parameters only. You might choose not to return a result set when your stored procedure creates or updates database tables. You can use the stored procedure wizards to define input and output parameters for a stored procedure so that it receives values for host variables from the client application. Additionally, you can create multiple SQL statements in a stored procedure, allowing the stored procedure to receive a case value and then to select one of a number of queries.

To build a stored procedure on a target database, simply click **Finish** in the stored procedure wizards. You do not have to manually register the stored procedure with DB2 by using the CREATE PROCEDURE statement.

Working with Existing Stored Procedures

After you successfully build a stored procedure on a database server, you are ready to modify, rebuild, run, and test the procedure. By modifying a stored procedure, you can add methods to the code to include sophisticated stored procedure logic. When you open a stored procedure in Stored Procedure Builder, the source code is displayed in the editor. The editor is language sensitive for stored procedures written in Java or SQL.

Running a stored procedure from within Stored Procedure Builder allows you to test the procedure to make sure that it is correctly installed. When you run a stored procedure, it can return result sets based on test input parameter values that you enter, depending on how you set up the stored procedure. Testing stored procedures makes programming the client application easier because you know that the stored procedure is correctly installed on the DB2 database server. You can then focus on writing and debugging the client application

From the Project window in Stored Procedure Builder, you can also easily drop a stored procedure or copy it to another database connection.

Creating Stored Procedure Builder Projects

When you open a new or existing Stored Procedure Builder project, the Project window shows all the stored procedures that reside on the DB2 database to which you are connected. You can choose to filter stored procedures to view the procedures based on their name or schema. A Stored Procedure Builder project saves only connection information and stored procedure objects that have not been successfully built to the database.

Debugging Stored Procedures

Using Stored Procedure Builder and the IBM Distributed Debugger (available separately), you can remotely debug a stored procedure installed on a DB2

server. To debug a stored procedure, you build the stored procedure in debug mode, add a debug entry for your client IP address, and run the stored procedure. You are not required to debug the stored procedures from within an application program. You can separate testing your stored procedure from testing the calling application program.

Using Stored Procedure Builder, you can view all the stored procedures that you have the authority to change, add, or remove debug entries for in the stored procedures debug table. If you are a database administrator or the creator of the selected stored procedure, you can grant authorization to other users to debug the stored procedure.

Part 4. Object-Relational Programming

Chapter 10. Using the Object-Relational Capabilities

Why Use the DB2 Object Extensions?

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. The external behavior and characteristics of objects are externalized whereas the internal implementation details of the object remain hidden. 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 behaviors, reflecting the behavior of your objects in the application domain.

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 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.

With the object extensions of DB2, you can incorporate object-oriented (OO) concepts and methodologies into your relational database.

Object-Relational Features of DB2

Some object-relational features that help you model your data in an object-oriented fashion include the following:

Data types for very large objects

The data you may need to model in your system may be very large and complex, such as text, audio, engineering data, or video. The VARCHAR or VARGRAPHIC data types may not be large enough for objects of this size. DB2 provides three data types to store these data objects as strings of up to 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).

User-defined data types

User-defined types let you control the semantics of your objects. For example, your application might require a type called "text" or a type called "address". These types do not exist as built-in types. However, with the object-relational features in DB2, you can define these types and use them in your database.

User-defined types can be further classified in the following ways:

Distinct types

Distinct types are based on existing DB2 built-in data types; that is, internally they are the same as built-in types, but you can define the semantics for those types. DB2 also has built-in types for storing and manipulating very large objects. Your distinct type could be based on one of these large object (LOB) data types, which you might want to use for something like an audio or video stream.

Structured types

Structured types are a way to gather together a collection of object attributes under a single type.

User-defined behaviors

You can write your own routines in SQL or an external language to enable DB2 to operate on your objects. There are two types of user-defined routines:

User-defined functions (UDFs)

UDFs are functions that you can define which, like built-in functions or operators, support the manipulation of objects in SQL queries. UDFs can be used to manipulate column values of any type, not just user-defined types.

User-defined methods

Like UDFs, methods define behavior for objects, but they are tightly encapsulated with a particular user-defined structured type.

Index extensions

Index extensions enable you to specify how DB2 indexes structured types and distinct types. To create an index extension, you must issue a CREATE INDEX EXTENSION statement. The CREATE INDEX EXTENSION statement specifies external table functions that convert values of a structured type or distinct type into index keys and define how DB2 searches through those index keys to optimize its performance.

For information on writing table functions, see "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393. For more information on using index extensions to improve the performance of

your applications that use structured types and distinct types, refer to the *Administration Guide*. For more information on the CREATE INDEX EXTENSION statement, refer to the *SQL Reference*.

Constraints

Constraints are rules that you define that the database enforces. There are four types of constraints:

Unique

Ensures the unique values of a key in a table. Any changes to the columns that compose the unique key are checked for uniqueness.

Referential integrity

Enforces referential constraints on insert, update, and delete operations. It is the state of a database in which all values of all foreign keys are valid.

Table check

Verify that changed data does not violate conditions specified when a table was created or altered.

Triggers

Triggers consist of SQL statements that are associated with a table and are automatically activated when data change operations occur on that table. You can use triggers to support general forms of integrity such as business rules.

For more information about unique constraints, referential integrity, and table check constraints, refer to the *Administration Guide*. For more information on triggers, refer to "Chapter 16. Using Triggers in an Active DBMS" on page 483.

Using object-oriented features in traditional applications

There is an important synergy among the object-oriented features of DB2. The use of the DB2 object-oriented mechanisms 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. The object-relational features of DB2 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.

User-defined Distinct Types

Distinct types are based on existing built-in types. For example, you might have distinct types to represent various currencies, such as USDollar and

Canadian_Dollar. Both of these types are represented internally (and in your host language program) as the built-in type that you defined these currencies on. For example, if you define both currencies as DECIMAL, they are represented as decimal data types in the system.

Strong typing

Although you can have different distinct types based on the same built-in type, distinct types have the property of *strong typing*. With this property of strong typing, you cannot directly compare instances of such types with anything other than another instance of that type. This prevents such semantically nonsensical operations such as directly adding USDollar and Canadian_Dollar without first going through a conversion process. You define which types of operations can occur for instances of a distinct type.

Type behavior

How do you define what operations are allowed on instances of USDollar or Canadian_Dollar? Use user-defined functions to define the allowable behaviors for instances of a distinct type. You can do something as simple as allowing instances of USDollar to be added together by registering a function that is really just the built-in addition operation that takes USDollar as input. You do not have to code an application to define this kind of function.

However, you may want to create a more complex function that can take the USDollar type as input and convert that to the Canadian_Dollar type. For more information about user-defined functions, refer to "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373.

You can implement integrity rules by using constraints.

Large objects

The objects you might model with distinct types might be very large. DB2 also has new built-in types for storing and manipulating very large objects. Your distinct type could be based on one of these large object (LOB) data types, which you might want to use for something like audio or video.

Defining Behavior for Objects: User-defined Routines

To define the behavior for your objects, you can use user-defined functions (UDFs) and methods:

User-defined functions

UDFs are functions that you can define which, like built-in functions or operators, support the manipulation of objects in SQL queries. (UDFs can be used to manipulate column values of any type, not just user-defined types.) Thus, instances of user-defined types (distinct or structured) are stored in columns or rows of tables and manipulated

by UDFs in SQL queries. For example, you might define a function AREA that takes an instance of the distinct type LENGTH and an instance of the distinct type WIDTH, computes the area, and returns it to the query:

```
SELECT ID, area(length, width) AS area
FROM Property
WHERE area > 10000;
```

Methods

Methods, like UDFs, define behavior for objects, but they differ from functions in the following ways:

- Methods are tightly associated with a particular user-defined structured type and are stored in the same schema as the user-defined type.
- Methods can be invoked on user-defined structured types that are stored as values in columns, or, using the dereference operator (->), on scoped references to structured types.
- Methods are invoked using a different SQL syntax from that used to invoke functions.
- DB2 resolves unqualified references to methods starting with the type on which the method was invoked. If the type on which the method was invoked does not define the method, DB2 tries to resolve the method by calling the method on the supertype of the type on which the method was invoked.

To invoke a method on a structured type stored in a column, include the name of the structured type (or an expression that resolves to a structured type), followed by the method invocation operator (..), followed by the name of the method. To invoke a method on a scoped reference of a structured type, include the reference to the structured type using the dereference operator (->), followed by the method invocation operator, followed by the name of the method.

For more information about the object-relational features of DB2, refer to:

- "Chapter 12. Working with Complex Objects: User-Defined Structured Types" on page 291
- "Chapter 11. User-defined Distinct Types" on page 281
- "Chapter 13. Using Large Objects (LOBs)" on page 349
- "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373
- "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393
- "Chapter 16. Using Triggers in an Active DBMS" on page 483

Chapter 11. User-defined Distinct Types

Why Use Distinct Types?

You can use data types that you have created, called *user-defined distinct types*, in your DB2 applications. There are several benefits associated with distinct types:

1. **Extensibility**.

   By defining new types, you can 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. For more information on UDFs, see "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373.

3. **Consistent behavior**.

   Strong typing insures that your distinct types will behave appropriately. It guarantees that only functions defined on your distinct type can be applied to instances of the distinct type.

4. **Encapsulation**.

   The set of functions and operators that you can apply to distinct types defines the behavior of your distinct types. This provides flexibility in the implementation since running applications do not depend on the internal representation that you choose for your type.

5. **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.

Defining a Distinct Type

Distinct types, 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 distinct type. 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 distinct type 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 distinct type is the type used by DB2 to internally represent the distinct type. For this reason, it must be a built-in data type. Previously defined distinct types cannot be used as source types of other distinct types.
3. The WITH COMPARISONS clause is used to tell DB2 that functions to support the comparison operations on instances of the distinct type 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 distinct type definition, DB2 always generates cast functions to:

- Cast from the distinct type 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 distinct type. Refer to the *SQL Reference* for a discussion of when additional casts to the distinct types are generated.

These functions are important for the manipulation of distinct types in queries.

Resolving Unqualified Distinct Types

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 387.

Examples of Using CREATE DISTINCT TYPE

The following are examples of using CREATE DISTINCT TYPE:

- Example: Money
- Example: Job Application

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 distinct types 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 EURO AS DECIMAL (9,2) WITH COMPARISONS
```

Note that you have to specify WITH COMPARISONS since comparison operators are supported on DECIMAL (9,2).

Example: Job Application

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 distinct type 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 distinct types and UDFs dealing with applicant forms in the same schema.

Defining Tables with Distinct Types

After you have defined several distinct types, you can start defining tables with columns whose types are distinct types. Following are examples using CREATE TABLE:

- Example: Sales
- Example: Application Forms

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         EURO)
```

The distinct types in the above examples are created using the same CREATE DISTINCT TYPE statements in "Example: Money" on page 283. Note that the above examples use check constraints. For information on check constraints refer to 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 distinct type 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 Distinct Types

One of the most important concepts associated with distinct types is *strong typing*. Strong typing guarantees that only functions and operators defined on the distinct type can be applied to its instances.

Strong typing is important to ensure that the instances of your distinct types 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 euros 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, distinct type instances with instances of the source type of the distinct type. For the same reason, DB2 will not let you apply functions defined on other types to distinct types. If you want to compare instances of distinct types 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 distinct type instance to the type of the parameter of a function that is not defined on a distinct type if you want to apply this function to a distinct type instance.

Examples of Manipulating Distinct Types

The following are examples of manipulating distinct types:

- Example: Comparisons Between Distinct Types and Constants
- Example: Casting Between Distinct Types
- Example: Comparisons Involving Distinct Types
- Example: Sourced UDFs Involving Distinct Types
- Example: Assignments Involving Distinct Types
- Example: Assignments in Dynamic SQL
- Example: Assignments Involving Different Distinct Types
- Example: Use of Distinct Types in UNION

Example: Comparisons Between Distinct Types and Constants

Suppose you want to know which products sold more than US $100 000.00 in the US in the month of July, 1999 (7/99).

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1999
```

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 distinct type, 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  = 1999
```

Example: Casting Between Distinct Types

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 distinct types 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 distinct type. 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 euros to U.S. dollars would be similar to the example above:

```
CREATE FUNCTION EURO_TO_US_DOUBL(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!euro2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE EURO_TO_US_DOUBL (DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

Example: Comparisons Involving Distinct Types

Suppose you want to know which products sold more in the US than in Canada and Germany for the month of July, 1999 (7/1999):

```
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 = 7
  AND US.YEAR  = 1999
```

```
  AND CDN.MONTH = 7
  AND CDN.YEAR  = 1999
  AND GERMAN.MONTH = 7
  AND GERMAN.YEAR  = 1999
```

Because you cannot directly compare US dollars with Canadian dollars or euros, you use the UDF to cast the amount in Canadian dollars to US dollars, and the UDF to cast the amount in euros 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 Distinct Types

Suppose you have defined a sourced UDF on the built-in SUM function to support SUM on euros:

```
CREATE FUNCTION SUM (EUROS)
  RETURNS EUROS
  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 Distinct Types

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 distinct type `personal.application_form` because DB2 lets you assign instances of the source type of a distinct type to targets having that distinct type.

Example: Assignments in Dynamic SQL

If you want to use the same statement given in "Example: Assignments Involving Distinct Types" on page 288 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 distinct type column. Remember that you cannot declare a host variable of a distinct type type, since host languages do not support distinct types. Therefore, you cannot specify that the type of a parameter marker is a distinct type.

Example: Assignments Involving Different Distinct Types

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 euros in "Example: Sourced UDFs Involving Distinct Types" on page 288:

```
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 euros to US dollars since different distinct types are not directly assignable to each other. You cannot use the cast specification syntax because distinct types can only be cast to their own source type.

Example: Use of Distinct Types 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 euros to US dollars because distinct types are union compatible only with the same distinct type. Note that you have to use the functional notation to cast between distinct types since the cast specification only lets you cast between distinct types and their source types.

Chapter 12. Working with Complex Objects: User-Defined Structured Types

Structured Types Overview

Structured types are useful for modelling objects that have a well-defined structure consisting of *attributes*. Attributes are properties that describe an instance of a type. A geometric shape, for example, might have as attributes its list of Cartesian coordinates. A person might have attributes of name, address, and so on. A department might have a name or some other kind of ID.

To create a type, you must specify the name of the type, its attribute names and their data types, and, optionally, how you want the reference type for this type to be represented in the system. Here is the SQL to create the `BusinessUnit_t` type:

```
CREATE TYPE BusinessUnit_t AS
  (Name VARCHAR(20),
  Headcount INT)
  REF USING INT
  MODE DB2SQL;
```

The AS clause provides the attribute definitions associated with the type. `BusinessUnit_t` is a type with two attributes: `Name` and `Headcount`. To create a structured type, you must include the MODE DB2SQL clause in the CREATE TYPE statement. For more information on the REF USING clause, see "Reference Types and Their Representation Types" on page 295.

Structured types offer two major extensions beyond traditional relational data types: the property of inheritance, and the capability of storing instances of a structured type either as rows in a table, or as values in a column. The following section briefly describes these features:

Inheritance

It is certainly possible to model objects such as people using traditional relational tables and columns. However, structured types offer an additional property of *inheritance*. That is, a structured type can have *subtypes* that reuse all of its attributes and contain additional attributes specific to the subtype. For example, the structured type `Person_t` might contain attributes for `Name`, `Age`, and `Address`. A subtype of `Person_t` might be `Employee_t`, that contains all of the attributes `Name`, `Age`, and `Address` and in addition contains attributes for `SerialNum`, `Salary`, and `BusinessUnit`.

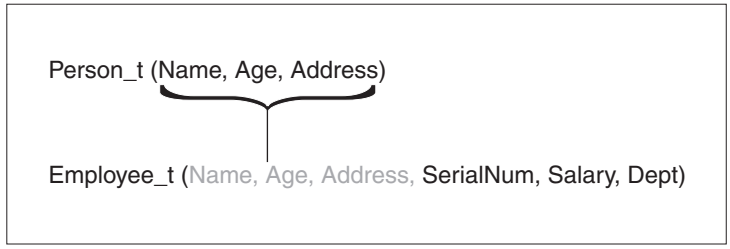


Figure 7. Structured type Employee_t inherits from Person_t

Storing instances of structured type

A structured type instance can be stored in the database in two ways:

- As a row in a table, in which each column of the table is an attribute of the instance of the type. To store objects as rows in a table, the table is defined with the structured type, rather than by specifying individual columns in the table definition:

  ```
  CREATE TABLE Person OF Person_t
     ...
  ```

  Each column in the table derives its name and data type from one of the attributes of the indicated structured type. Such tables are known as typed tables.
- As a value in a column. To store objects in table columns, the column is defined using the structured type as its type. The following statement creates a `Properties` table that has a structured type `Address` that is of the `Address_t` structured type:

  ```
  CREATE TABLE Properties
     (ParcelNum INT,
     Photo BLOB(2K),
     Address Address_t)
     ...
  ```

Creating a Structured Type Hierarchy

A structured type may be created *under* another structured type, in which case the newly created type is a *subtype* of the original structured type. The original type is the *supertype*. The subtype inherits all the attributes of the supertype, and can optionally have additional attributes of its own.

For example, a data model may need to represent a special type of employee called a manager. Managers have more attributes than employees who are not managers. The `Manager_t` type inherits the attributes defined for an employee, but also is defined with some additional attributes of its own, such as a special bonus attribute that is only available to managers. The type hierarchies used for examples in this book are shown in Figure 8 on page 294. The type

hierarchy for Address_t is defined in "Inserting Structured Type Instances into a Column" on page 321.

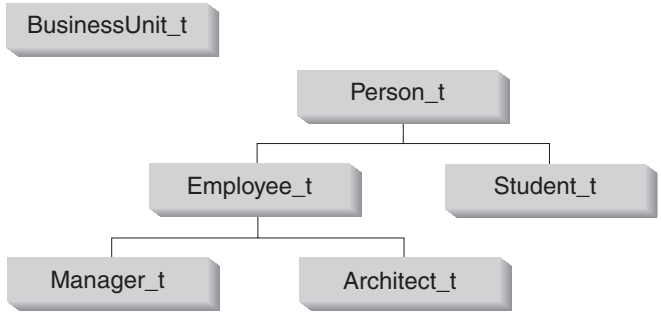


Figure 8. Type hierarchies (BusinessUnit_t and Person_t)

In Figure 8, the person type Person_t is the *root type* of the hierarchy. Person_t is also the supertype of the types below it--in this case, the type named Employee_t and the type named Student_t. The relationships among subtypes and supertypes are transitive; in other words, the relationship between subtype and supertype exists throughout the entire type hierarchy. So, Person_t is also a supertype of types Manager_t and Architect_t.

Type BusinessUnit_t, defined in "Structured Types Overview" on page 292, has no subtypes. Type Address_t, defined in "Inserting Structured Type Instances into a Column" on page 321, has the following subtypes: Germany_addr_t, Brazil_addr_t, and US_addr_t.

The CREATE TYPE statement for type Person_t declares that Person_t is INSTANTIABLE. For more information on declaring structured types using the INSTANTIABLE or NOT INSTANTIABLE clauses, see "Additional Properties of Structured Types" on page 303.

The following SQL statements create the Person_t type hierarchy:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   INSTANTIABLE
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;

CREATE TYPE Employee_t UNDER Person_t AS
   (SerialNum INT,
   Salary DECIMAL (9,2),
   Dept REF(BusinessUnit_t))
   MODE DB2SQL;
```

```
CREATE TYPE Student_t UNDER Person_t AS
   (SerialNum CHAR(6),
   GPA DOUBLE)
   MODE DB2SQL;

CREATE TYPE Manager_t UNDER Employee_t AS
   (Bonus DECIMAL (7,2))
   MODE DB2SQL;

CREATE TYPE Architect_t UNDER Employee_t AS
   (StockOption INTEGER)
   MODE DB2SQL;
```

Person_t has three attributes: Name, Age and Address. Its two subtypes, Employee_t and Student_t, each inherit the attributes of Person_t and also have several additional attributes that are specific to their particular types. For example, although both employees and students have serial numbers, the format used for student serial numbers is different from the format used for employee serial numbers.

Note: A typed table created from the Person_t type includes the column Address of structured type Address_t. As with any structured type column, you must define transform functions for the structured type of that column. For information on defining transform functions, see "Creating the Mapping to the Host Language Program: Transform Functions" on page 329.

Finally, Manager_t and Architect_t are both subtypes of Employee_t; they inherit all the attributes of Employee_t and extend them further as appropriate for their types. Thus, an instance of type Manager_t will have a total of seven attributes: Name, Age, Address, SerialNum, Salary, Dept, and Bonus.

Reference Types and Their Representation Types

For every structured type you create, DB2 automatically creates a companion type. The companion type is called a *reference type* and the structured type to which it refers is called a *referenced type*. Typed tables can make special use of the reference type, as described in "Using Structured Types in Typed Tables" on page 304. You can also use reference types in SQL statements like other user-defined types. To use a reference type in an SQL statement, use REF(*type-name*), where *type-name* represents the referenced type.

DB2 uses the reference type as the type of the object identifier column in typed tables. The object identifier uniquely identifies a row object in the typed table hierarchy. DB2 also uses reference types to store references to rows in typed tables. You can use reference types to refer to each row object in the table. For more information about using references, see "Using Reference Types" on page 308. For more information on typed tables, see "Storing Objects in Typed Tables" on page 299.

References are strongly typed. Therefore, you must have a way to use the type in expressions. When you create the root type of a type hierarchy, you can specify the base type for a reference with the REF USING clause of the CREATE TYPE statement. The base type for a reference is called the *representation type*. If you do not specify the representation type with the REF USING clause, DB2 uses the default data type of VARCHAR(16) FOR BIT DATA. The representation type of the root type is inherited by all its subtypes. The REF USING clause is only valid when you define the root type of a hierarchy. In the examples used throughout this section, the representation type for the `BusinessUnit_t` type is INTEGER, while the representation type for `Person_t` is VARCHAR(13).

Casting and Comparing Reference Types

DB2 automatically creates functions that cast values between the reference type and its representation type, in both directions. The CREATE TYPE statement has an optional CAST WITH clause, described in the *SQL Reference*, that allows you to choose the names of these two cast functions. By default, the names of the cast functions are the same as the names of the structured type and its reference representation type. For example, the CREATE TYPE Person_t statement from "Creating a Structured Type Hierarchy" on page 293 automatically creates the following functions:

```
CREATE FUNCTION VARCHAR(REF(Person_t))
   RETURNS VARCHAR
```

DB2 also creates the function that does the inverse operation:

```
CREATE FUNCTION Person_t(VARCHAR(13))
   RETURNS REF(Person_t)
```

You will use these cast functions whenever you need to insert a new value into the typed table or when you want to compare a reference value to another value.

DB2 also creates functions that let you compare reference types using the following comparison operators: `=`, `<>`, `<`, `<=`, `>`, and `>=`. For more information on comparison operators for reference types, refer to the *SQL Reference*.

Other System-Generated Routines

Every structured type that you create causes DB2 to implicitly create a set of functions and methods that you can use to construct, observe, or modify a structured type value. This means, for instance, that for type `Person_t`, DB2 automatically creates the following functions and methods when you create the type:

Constructor function
: A function of the same name as the type is created. This function has no parameters and returns an instance of the type with all of its

attributes set to null. The function that is created for Person_t, for example, is as if the following statement were executed:

```
CREATE FUNCTION Person_t ( ) RETURNS Person_t
```

For the subtype Manager_t, a constructor function is created as if the following statement had been executed:

```
CREATE FUNCTION Manager_t ( ) RETURNS Manager_t
```

To construct an instance of a type to insert into a column, use the constructor function with the mutator methods. If the type is stored in a table, rather than a column, you do not have to use the constructor function with the mutator methods to insert an instance of a type. For more information on inserting data into typed tables, see "Inserting Rows that Contain Structured Type Values" on page 323.

Mutator methods

A mutator method exists for each attribute of an object. The instance of a type on which a method is invoked is called the *subject* instance of the method. When the mutator method invoked on a subject instance receives a new value for an attribute, the method returns a new instance with the attribute updated to the new value. So, for type Person_t, DB2 creates mutator methods for each of the following attributes: name, age, and address.

The mutator method DB2 creates for attribute age, for example, is as if the following statement had been executed:

```
ALTER TYPE Person_t
   ADD METHOD AGE(int)
   RETURNS Person_t;
```

For more information on mutating objects, see "Retrieving and Modifying Structured Type Values" on page 324.

Observer methods

An observer method exists for each attribute of an object. If the method for an attribute receives an object of the expected type or subtype, the method returns the value of the attribute for that object.

The observer method DB2 creates for the attribute age of the type Person_t, for example, is as if DB2 issued the following statement:

```
ALTER TYPE Person_t
   ADD METHOD AGE()
   RETURNS INTEGER;
```

For more information about using observer methods, see "Retrieving and Modifying Structured Type Values" on page 324.

To invoke a method on a structured type, use the method invocation operator: '..'. For more information about method invocation, refer to the *SQL Reference*.

Defining Behavior for Types

To define behaviors for structured types, you can create user-defined methods. You cannot create methods for distinct types. Creating a method is similar to creating a function, with the exception that methods are created specifically for a type, so that the type and its behavior are tightly integrated.

The method specification must be associated with the type before you issue the CREATE METHOD statement. The following statement adds the method specification for a method called `calc_bonus` to the `Employee_t` type:

```
ALTER TYPE Employee_t
   ADD METHOD calc_bonus (rate DOUBLE)
   RETURNS DECIMAL(7,2)
   LANGUAGE SQL
   CONTAINS SQL
   NO EXTERNAL ACTION
   DETERMINISTIC;
```

Once you have associated the method specification with the type, you can define the behavior for the type by creating the method as either an external method or an SQL-bodied method, according to the method specification. For example, the following statement registers an SQL method called `calc_bonus` that resides in the same schema as the type `Employee_t`:

```
CREATE METHOD calc_bonus (rate DOUBLE)
   FOR Employee_t
   RETURN SELF..salary * rate;
```

You can create as many methods named `calc_bonus` as you like, as long as they have different numbers or types of parameters, or are defined for types in different type hierarchies. In other words, you cannot create another method named `calc_bonus` for `Architect_t` that has the same parameter types and same number of parameters.

Note: DB2 does not currently support dynamic dispatch. This means that you cannot declare a method for a type, and then redefine the method for a subtype using the same number of parameters. As a workaround, you can use the TYPE predicate to determine the dynamic type and then use the TREAT AS clause to call a different method for each dynamic type. For an example of transform functions that handle subtypes, see "Retrieving Subtype Data from DB2 (Bind Out)" on page 341.

For more information about registering, writing, and invoking methods, see "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373 and "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393.

Storing Objects in Typed Tables

You can store instances of structured types either as rows in typed tables, in which each attribute of the type is stored in a separate column, or as objects in columns, in which all of the attributes of the type are stored in a single column. Typed tables have the attribute of identity; that is, another table can use references to access attributes of instances. If you need to refer to your instance from other tables, you must use typed tables. If your objects do not need to be identified by other tables, consider storing the objects in columns.

When objects are stored as rows in a table, each column of the table contains one attribute of the object. You could store an instance of a person, for example, in a table that contains a column for name and a column for age. Here is an example of a CREATE TABLE statement for storing instances of `Person`.

```
CREATE TABLE Person OF Person_t
   (REF IS Oid USER GENERATED)
```

To insert an instance of `Person` into the table, you could use the following syntax:

```
INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 29);
```

Table 10. Person typed table

Oid	Name	Age	Address
a	Andrew	29	

Your program accesses attributes of the object by accessing the columns of the typed table:

UPDATE Person SET Age=30 WHERE Name='Andrew';

After the previous UPDATE statement, the table looks like:

Table 11. Person typed table after update

Oid	Name	Age	Address
a	Andrew	30	

Because there is a subtype of `Person_t` called `Employee_t`, instances of `Employee_t` cannot be stored in the `Person` table, and need to be stored in another table. This table is called a *subtable*. The following CREATE TABLE statement creates the `Employee` subtable under the `Person` table:

```
CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit);
```

And, again, an insert into the Employee table looks like this:

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary)
   VALUES (Employee_t('s'), 'Susan', 39, 24001, 37000.48)
```

Table 12. Employer typed subtable

Oid	Name	Age	Address	SerialNum	Salary	Dept
s	Susan	39		24001	37000.48	

If you execute the following query, the information for Susan is returned:

```
SELECT *
   FROM Employee
   WHERE Name='Susan';
```

The interesting thing about these two tables is that you can access instances of both employees and people just by executing your SQL statement on the Person table. This feature is called *substitutability*, and is discussed in "Additional Properties of Structured Types" on page 303. By executing a query on the table that contains instances that are higher in the type hierarchy, you automatically get instances of types that are lower in the hierarchy. In other words, the Person table logically looks like this to SELECT, UPDATE, and DELETE statements :

Table 13. Person table contains Person and Employee instances

Oid	Name	Age	Address
a	Andrew	30	(null)
s	Susan	39	(null)

If you execute the following query, you get an object identifier and Person_t information about both Andrew (a person) and Susan (an employee):

```
SELECT *
   FROM Person;
```

For more information on substitutability, see "Additional Properties of Structured Types" on page 303.

Defining Relationships Between Objects in Typed Tables

You can define relationships between objects in one typed table and objects in another table. You can also define relationships between objects in the same typed table. For example, assume that you have defined a typed table that

contains instances of departments. Instead of maintaining department numbers in the Employee table, the Dept column of the Employee table can contain a logical pointer to one of the departments in the BusinessUnit table. These pointers are called *references*, and are illustrated in Figure 9.

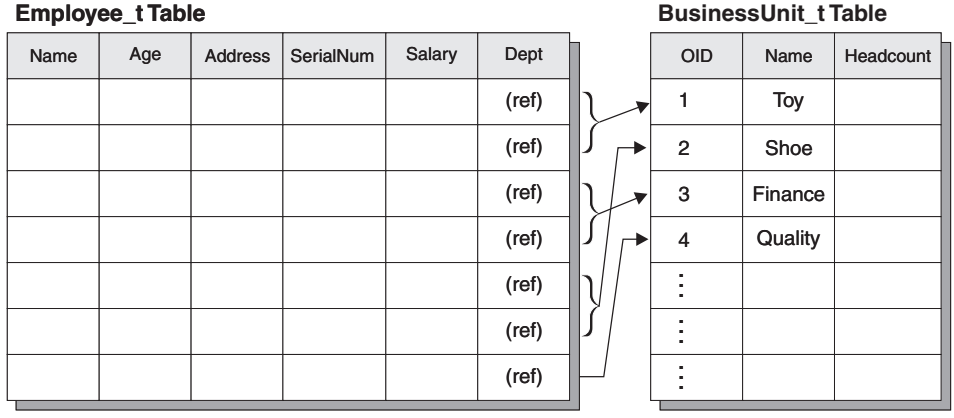


Figure 9. Structured type references from Employee_t to BusinessUnit_t

Important: References do not perform the same function as referential constraints. It is possible to have a reference to a department that does not exist. If it is important to maintain integrity between department and employees, you can define a referential constraint between those two tables. The real power of references is that it gives you the ability to write queries that navigate the relationship between the tables. What the query does is dereference the relationship and instantiate the object that is being pointed to. The operator that you use to perform this action is called the *dereference* operator, which looks like this: ->.

For example, the following query on the Employee table uses the dereference operator to tell DB2 to follow the path from the Dept column to the BusinessUnit table. The dereference operator returns the value of the Name column:

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

For more information on writing queries on typed tables, see "Querying a Typed Table" on page 314.

Storing Objects in Columns

Storing objects in columns is useful when you need to model facts about your business objects that cannot be adequately modelled with the DB2 built-in data types. In other words, you may store your business objects (such as

employees, departments, and so on) in typed tables, but those objects might also have attributes that are best modelled using a structured type.

For example, assume that your application has the need to access certain parts of an address. Rather than store the address as an unstructured character string, you can store it as a structured object as shown in Figure 10.

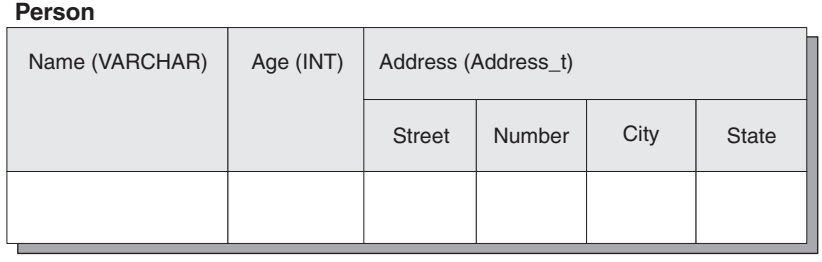


Figure 10. Address attribute as a structured type

Furthermore, you can define a type hierarchy of addresses to model different formats of addresses that are used in different countries. For example, you might want to include both a US address type, which contains a zip code, and a Brazilian address type, for which the neighborhood attribute is required. The `Address_t` type hierarchy is defined in "Inserting Structured Type Instances into a Column" on page 321.

When objects are stored as column values, the attributes are not externally represented as they are with objects stored in rows of tables. Instead, you must use methods to manipulate their attributes. DB2 generates both *observer* methods to return attributes, and *mutator* methods to change attributes. The following examples uses one observer method and two mutator methods, one for the `Number` attribute and one for the `Street` attribut, to change an address:

```
UPDATE Employee
   SET Address=Address..Number('4869')..Street('Appletree')
   WHERE Name='Franky'
   AND Address..State='CA';
```

In the preceding example, the SET clause of the UPDATE statement invokes the `Number` and `Street` mutator methods to update attributes of the instances of type `Address_t`. The WHERE clause restricts the operation of the update statement with two predicates: an equality comparison for the `Name` column, and an equality comparison that invokes the `State` observer method of the `Address` column.

Additional Properties of Structured Types

Substitutability

When a SELECT, UPDATE, or DELETE statement is applied to a typed table, the operation applies to the named table and all of its subtables. For example, if you create a typed table from `Person_t` and select all rows from that table, your application can receive not just instances of the `Person` type, but `Person` information about instances of the `Employee` subtype and other subtypes. The property of substitutability also applies to subtables created from subtypes. For example, SELECT, UPDATE, and DELETE statements for the `Employee` subtable apply to both the `Employee_t` type and its own subtypes.

Similarly, a column defined with `Address_t` type can contain instances of a US address or a Brazilian address.

INSERT operations, in contrast, only apply to the table that is specified in the INSERT statement. Inserting into the `Employee` table creates an `Employee_t` object in the `Person` table hierarchy.

You can also substitute subtype instances when you pass structured types as parameters to functions, or as the result from a function. If a scalar function has a parameter of type `Address_t`, you can pass an instance of one of its subtypes, such as `US_addr_t`, instead of an instance of `Address_t`. Table functions cannot return structured type columns.

Because a column or table is defined with one type but might contain instances of other types, it is sometimes important to distinguish between the type that was used for the definition and the type of the instance that is actually returned at runtime. The definition of the structured type in a column, row, or function parameter is called the *static type*. The actual type of a structured type instance is called the *dynamic type*.To retrieve information about the dynamic type, your application can use the TYPE_NAME, TYPE_SCHEMA, and TYPE_ID built-in functions that are described in "Other Type-related Built-in Functions" on page 316.

Instantiability

Types can also be defined to be *INSTANTIABLE* or *NOT INSTANTIABLE*. By default, types are instantiable, which means that an instance of that object can be created. Noninstantiable types, on the other hand, serve as models intended for further refinement in the type hierarchy. For example, if you define `Person_t` using the NOT INSTANTIABLE clause, then you cannot store any instances of a person in the database, and you cannot create a table or view using `Person_t`. Instead, you can only store instances of `Employee_t` or other subtypes of `Person_t` that you define.

Using Structured Types in Typed Tables

Creating a Typed Table

Typed tables are used to actually store instances of objects whose characteristics are defined with the CREATE TYPE statement. You can create a typed table using a variant of the CREATE TABLE statement. You can also create a hierarchy of typed tables that is based on a hierarchy of structured types. To store instances of subtypes in database tables, you must create a corresponding table hierarchy.

The following example illustrates creation of a table hierarchy based on the type hierarchy shown in Figure 9 on page 301.

Here is the SQL to create the `BusinessUnit` typed table:

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
   (REF IS Oid USER GENERATED);
```

Here is the SQL to create the tables in the `Person` table hierarchy:

```
CREATE TABLE Person OF Person_t
   (REF IS Oid USER GENERATED);

CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit );

CREATE TABLE Student OF Student_t UNDER Person
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Manager OF Manager_t UNDER Employee
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Architect OF Architect_t UNDER Employee
   INHERIT SELECT PRIVILEGES;
```

Defining the Type of the Table

The first typed table created in the previous example is `BusinessUnit`. This table is defined to be OF type `BusinessUnit_t`, so it will hold instances of that type. This means that it will have a column corresponding to each attribute of the structured type `BusinessUnit_t`, and one additional column called the *object identifier column*.

Naming the Object Identifier

Because typed tables contain objects that can be referenced by other objects, every typed table has an *object identifier* column as its first column. In this example, the type of the object identifier column is REF(`BusinessUnit_t`). You can name the object identifier column using the REF IS ... USER GENERATED clause. In this case, the column is named `Oid`. The USER GENERATED part of

the REF IS clause indicates that you must provide the initial value for the object identifier column of each newly inserted row. After you insert the object identifier, you cannot update the value of the object identifier. For information on configuring DB2 to automatically generate object identifiers, see "Defining System-generated Object Identifiers" on page 319.

Specifying the Position in the Table Hierarchy

The `Person` typed table is of type `Person_t`. To store instances of the subtypes of employees and students, it is necessary to create the subtables of the `Person` table, `Employee` and Student. The two additional subtypes of `Employee_t` also require tables. Those subtables are named `Manager` and `Architect`. Just as a subtype inherits the attributes of its supertype, a subtable inherits the columns of its supertable, including the object identifier column.

Note: A subtable must reside in the same schema as its supertable.

Rows in the `Employee` subtable, therefore, will have a total of seven columns: `Oid`, `Name`, `Age`, `Address`, `SerialNum`, `Salary`, and `Dept`.

A SELECT, UPDATE, or DELETE statement that operates on a supertable automatically operates on all its subtables as well. For example, an UPDATE statement on the `Employee` table might affect rows in the `Employee`, `Manager`, and `Architect` tables, but an UPDATE statement on the `Manager` table can only affect `Manager` rows.

If you want to restrict the actions of the SELECT, INSERT, or DELETE statement to just the specified table, use the ONLY option, described in "Returning Objects of a Particular Type Using ONLY" on page 317.

Indicating that SELECT Privileges are Inherited

The INHERIT SELECT PRIVILEGES clause of the CREATE TABLE statement specifies that the resulting subtable, such as `Employee`, is initially accessible by the same users and groups as the supertable, such as `Person`, from which it is created using the UNDER clause. Any user or group currently holding SELECT privileges on the supertable is granted SELECT privileges on the newly created subtable. The creator of the subtable is the grantor of the SELECT privileges. To specify privileges such as DELETE and UPDATE on subtables, you must issue the same explicit GRANT or REVOKE statements that you use to specify privileges on regular tables. For more information on the INHERIT SELECT PRIVILEGES clause, refer to the *SQL Reference*.

Privileges may be granted and revoked independently at every level of a table hierarchy. If you create a subtable, you can also revoke the inherited SELECT privileges on that subtable. Revoking the inherited SELECT privileges from the subtable prevents users with SELECT privileges on the supertable from seeing any columns that appear only in the subtable. Revoking the inherited

SELECT privileges from the subtable limits users who only have SELECT privileges on the supertable to seeing the supertable columns of the rows of the subtable. Users can only operate directly on a subtable if they hold the necessary privilege on that subtable. So, to prevent users from selecting the bonuses of the managers in the subtable, revoke the SELECT privilege on that table and grant it only to those users for whom this information is necessary.

Defining Column Options

The WITH OPTIONS clause lets you define options that apply to an individual column in the typed table. The format of WITH OPTIONS is:

```
column-name WITH OPTIONS column-options
```

where *column-name* represents the name of the column in the CREATE TABLE or ALTER TABLE statement, and *column-options* represents the options defined for the column.

For example, to prevent users from inserting nulls into a `SerialNum` column, specify the NOT NULL column option as follows:

```
(SerialNum WITH OPTIONS NOT NULL)
```

Defining the Scope of a Reference Column

Another use of WITH OPTIONS is to specify the SCOPE of a column. For example, in the `Employee` table and its subtables, the clause:

```
Dept WITH OPTIONS SCOPE BusinessUnit
```

declares that the `Dept` column of this table and its subtables have a *scope* of `BusinessUnit`. This means that the reference values in this column of the `Employee` table are intended to refer to objects in the `BusinessUnit` table.

For example, the following query on the `Employee` table uses the dereference operator to tell DB2 to follow the path from the `Dept` column to the `BusinessUnit` table. The dereference operator returns the value of the `Name` column:

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

For more information about references and scoping references, see "Using Reference Types" on page 308.

Populating a Typed Table

After creating the structured types in the previous examples, and after creating the corresponding tables and subtables, the structure of your database looks like Figure 11 on page 307:

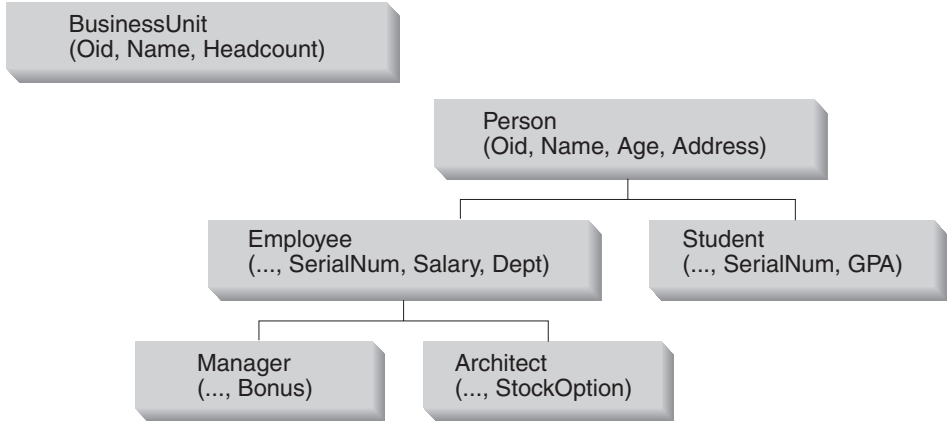


Figure 11. Typed table hierarchy

When the hierarchy is established, you can use the INSERT statement, as usual, to populate the tables. The only difference is that you must remember to populate the object identifier columns and, optionally, any additional attributes of the objects in each table or subtable. Because the object identifier column is a REF type, which is strongly typed, you must cast the user-provided object identifier values, using the cast function that the system generated for you when you created the structured type.

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(1), 'Toy', 15);

INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(2), 'Shoe', 10);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 20);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('b'), 'Bob', 30);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('c'), 'Cathy', 25);

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000, BusinessUnit_t(1));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('e'), 'Eva', 31, 83, 45000, BusinessUnit_t(2));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('f'), 'Franky', 28, 214, 39000, BusinessUnit_t(2));

INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('g'), 'Gordon', 19, '10245', 4.7);
```

```
INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('h'), 'Helen', 20, '10357', 3.5);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('i'), 'Iris', 35, 251, 55000, BusinessUnit_t(1), 12000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept,
Bonus)
   VALUES(Manager_t('j'), 'Christina', 10, 317, 85000, BusinessUnit_t(1),
25000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('k'), 'Ken', 55, 482, 105000, BusinessUnit_t(2), 48000);

INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('l'), 'Leo', 35, 661, 92000, BusinessUnit_t(2), 20000);
```

The previous example does not insert any addresses. For information about how to insert structured type values into columns, see "Inserting Rows that Contain Structured Type Values" on page 323.

When you insert rows into a typed table, the first value in each inserted row must be the object identifier for the data being inserted into the tables. Also, just as with non-typed tables, you must provide data for all columns that are defined as NOT NULL. Finally, notice that any reference-valued expression of the appropriate type can be used to initialize a reference attribute. In the previous examples, the `Dept` reference of the employees is input as an appropriately type-cast constant. However, you can also obtain the reference using a subquery, as shown in the following example:

```
INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('m'), 'Brian', 7, 882, 112000,
   (SELECT Oid FROM BusinessUnit WHERE name = 'Toy'), 30000);
```

Using Reference Types

For each structured type, DB2 supports a corresponding reference type. For example, when you create the `Person_t` type, DB2 automatically creates a type of REF(Person_t). The representation type of the REF(Person_t) type (and the REF types of all subtypes of `Person_t`) is, by default, VARCHAR (16) FOR BIT DATA, but you can choose a different representation type using the REF USING clause for the CREATE TYPE statement. That reference type is the basis of the object identifier column of the typed table that you create to store instances of the structured type. For example, if you create a root type `People_t` using the default representation type for the reference type, the object identifier column of the associated `People` table is based on VARCHAR(16) FOR BIT DATA.

Comparing Reference Types

Reference types are strongly typed. To compare a reference to a constant, you can cast the constant to the appropriate reference type, or you can cast the

reference type to the base type, and then perform the comparison. All references in a given type hierarchy have the same reference representation type. This enables REF(S) and REF(T) to be compared, provided that S and T have a common supertype. Because uniqueness of the object identifier column is enforced only within a table hierarchy, it is possible that a value of REF(T) in one table hierarchy may be equal to a value of REF(T) in another table hierarchy, even though they reference different rows.

Using References to Define Semantic Relationships

Using the WITH OPTIONS clause of CREATE TABLE, you can define that a relationship exists between a column in one table and the objects in the same or another table. For example, in the `BusinessUnit` and `Person` table hierarchies, the department for each employee is actually a reference to an object in the `BusinessUnit` table, as shown in Figure 12. To define the destination objects of a given reference column, use the SCOPE keyword on the WITH OPTIONS clause.

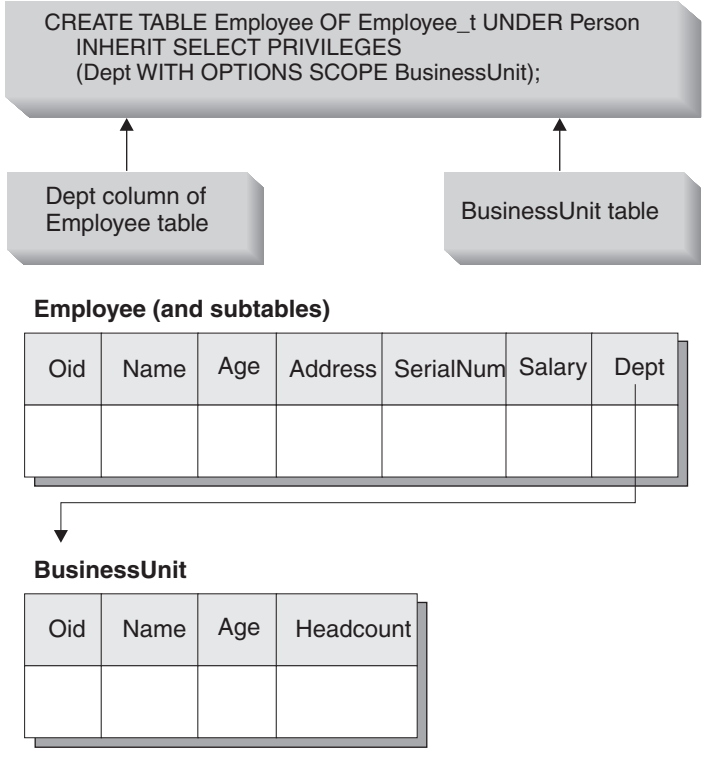


Figure 12. Dept attribute refers to a BusinessUnit object

Self-Referencing Relationships: You can define scoped references to objects in the same typed table as well. The statements in the following example

create one typed table for parts and one typed table for suppliers. To show the reference type definitions, the sample also includes the statements used to create the types:

```
CREATE TYPE Company_t AS
   (name VARCHAR(30),
   location VARCHAR(30))
   MODE DB2SQL ;

CREATE TYPE Part_t AS
   (Descript VARCHAR(20),
   Supplied_by REF(Company_t),
   Used_in REF(part_t))
   MODE DB2SQL;

CREATE TABLE Suppliers OF Company_t
   (REF IS suppno USER GENERATED);

CREATE TABLE Parts OF Part_t
   (REF IS Partno USER GENERATED,
   Supplied_by WITH OPTIONS SCOPE Suppliers,
   Used_in WITH OPTIONS SCOPE Parts);
```

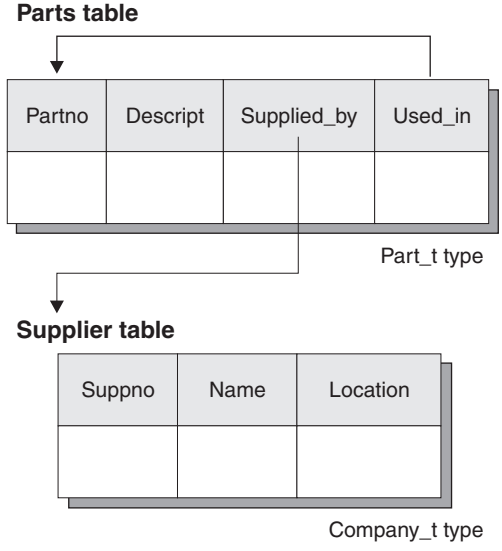


Figure 13. Example of a self-referencing scope

You can use scoped references to write queries that, without scoped references, would have to be written as outer joins or correlated subqueries. For more information, see "Queries that Dereference References" on page 315.

Differences Between Referential Integrity and Scoped References

Although scoped references do define relationships among objects in tables, they are different than referential integrity relationships. Scopes simply provide information about a target table. That information is used when dereferencing objects from that target table. Scoped references do not require or enforce that a value exists at the other table. For example, the `Dept` column in the `Employee` table can have a reference to a `BusinessUnit` object identifier column that does not exist in the `BusinessUnit` table. To ensure that the objects in these relationships exist, you must add a referential constraint between the tables. For more information, see "Creating Constraints on Object Identifier Columns" on page 320.

Creating a Typed View

You can create a typed view using the CREATE VIEW statement. For example, to create a view of the typed `BusinessUnit` table, you can define a structured type that has the desired attributes and then create a typed view using that type:

```
CREATE TYPE VBusinessUnit_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE VIEW VBusinessUnit OF VBusinessUnit_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VBusinessUnit_t(VARCHAR(Oid)), Name FROM BusinessUnit;
```

The OF clause in the CREATE VIEW statement tells DB2 to base the columns of the view on the attributes of the indicated structured type. In this case, DB2 bases the columns of the view on the `VBusinessUnit_t` structured type.

The `VObjectID` column of the view has a type of REF(VBusinessUnit_t). Since you cannot cast from a type of REF(BusinessUnit_t) to REF(VBusinessUnit_t), you must first cast the value of the `Oid` column from table `BusinessUnit` to data type VARCHAR, and then cast from data type VARCHAR to data type REF(VBusinessUnit_t).

The MODE DB2SQL clause specifies the mode of the typed view. This is the only valid mode currently supported.

The REF IS... clause is identical to that of the typed CREATE TABLE statement. It provides a name for the object identifier column of the view (`VObjectID` in this case), which is the first column of the view. If you create a typed view on a root type, you must specify an object identifier column for the view. If you create a typed view on a subtype, your view can inherit the object identifier column.

The USER GENERATED clause specifies that the initial value for the object identifier column must be provided by the user when inserting a row. Once inserted, the object identifier column cannot be updated.

The *body* of the view, which follows the keyword AS, is a SELECT statement that determines the content of the view. The column-types returned by this SELECT statement must be compatible with the column-types of the typed view, including the initial object identifier column.

To illustrate the creation of a typed view hierarchy, the following example defines a view hierarchy that omits some sensitive data and eliminates some type distinctions from the Person table hierarchy created earlier under "Creating a Typed Table" on page 304:

```
CREATE TYPE VPerson_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE TYPE VEmployee_t UNDER VPerson_t
   AS (Salary INT, Dept REF(VBusinessUnit_t))
   MODE DB2SQL;

CREATE VIEW VPerson OF VPerson_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VPerson_t (VARCHAR(Oid)), Name FROM ONLY(Person);

CREATE VIEW VEmployee OF VEmployee_t MODE DB2SQL
   UNDER VPerson INHERIT SELECT PRIVILEGES
   (Dept WITH OPTIONS SCOPE VBusinessUnit)
   AS SELECT VEmployee_t(VARCHAR(Oid)), Name, Salary,
      VBusinessUnit_t(VARCHAR(Dept))
   FROM Employee;
```

The two CREATE TYPE statements create the structured types that are needed to create the object view hierarchy for this example.

The first typed CREATE VIEW statement above creates the root view of the hierarchy, VPerson, and is very similar to the VBusinessUnit view definition. The difference is the use of ONLY(Person) to ensure that only the rows in the Person table hierarchy that are in the Person table, and not in any subtable, are included in the VPerson view. This ensures that the Oid values in VPerson are unique compared with the Oid values in VEmployee. The second CREATE VIEW statement creates a subview VEmployee under the view VPerson. As was the case for the UNDER clause in the CREATE TABLE...UNDER statement, the UNDER clause establishes the view hierarchy. You must create a subview in the same schema as its superview. Like typed tables, subviews inherit columns from their superview. Rows in the VEmployee view inherit the columns VObjectID and Name from VPerson and have the additional columns Salary and Dept associated with the type VEmployee_t.

The INHERIT SELECT PRIVILEGES clause has the same effect when you issue a CREATE VIEW statement as when you issue a typed CREATE TABLE statement. For more information on the INHERIT SELECT PRIVILEGES clause, see "Indicating that SELECT Privileges are Inherited" on page 305. The

WITH OPTIONS clause in a typed view definition also has the same effect as it does in a typed table definition. The WITH OPTIONS clause enables you to specify column options such as SCOPE. The READ ONLY clause forces a superview column to be marked as read-only, so that subsequent subview definitions can specify an expression for the same column that is also read-only.

If a view has a reference column, like the `Dept` column of the `VEmployee` view, you must associate a scope with the column to use the column in SQL dereference operations. If you do not specify a scope for the reference column of the view and the underlying table or view column is scoped, then the scope of the underlying column is passed on to the reference column of the view. You can explicitly assign a scope to the reference column of the view by using the WITH OPTIONS clause. In the previous example, the `Dept` column of the `VEmployee` view receives the `VBusinessUnit` view as its scope. If the underlying table or view column does not have a scope, and no scope is explicitly assigned in the view definition, or no scope is assigned with an ALTER VIEW statement, the reference column remains unscoped.

There are several important rules associated with restrictions on the queries for typed views found in the *SQL Reference* that you should read carefully before attempting to create and use a typed view.

Dropping a User-Defined Type (UDT) or Type Mapping

You can drop a user-defined type (UDT) or type mapping using the DROP statement. For more information on type mappings, see "Working with Data Type Mappings" on page 579. You cannot drop a UDT if it is used:

- In a column definition for an existing table or view.
- As the type of an existing typed table or typed view (structured type).
- As the supertype of another structured type.

You cannot drop a default type mapping; you can only override it by creating another type mapping.

The database manager attempts to drop every user-defined function (UDF) that is dependent on this UDT. A UDF cannot be dropped if a view, trigger, table check constraint, or another UDF is dependent on it. If DB2 cannot drop a dependent UDF, DB2 does not drop the UDT. Dropping a UDT invalidates any packages or cached dynamic SQL statements that used it.

If you have created a transform for a UDT, and you plan to drop that UDT, consider dropping the associated transform. To drop a transform, issue a DROP TRANSFORM statement. For the complete syntax of the DROP

TRANSFORM statement, refer to the *SQL Reference*. Note that you can only drop user-defined transforms. You cannot drop built-in transforms or their associated group definitions.

Altering or Dropping a View

The ALTER VIEW statement modifies an existing view by altering a reference type column to add a scope. Any other changes you make to a view require that you drop and then re-create the view.

When altering the view, the scope must be added to an existing reference type column that does not already have a scope defined. Further, the column must not be inherited from a superview.

The data type of the column-name in the ALTER VIEW statement must be REF (type of the typed table name or typed view name).

Refer to the *SQL Reference* for additional information on the ALTER VIEW statement.

The following example shows how to drop the EMP_VIEW:

```
DROP VIEW EMP_VIEW;
```

Any views that are dependent on the dropped view become inoperative. For more information on inoperative views, refer to the "Recovering Inoperative Views" section of the *Administration Guide*.

Other database objects such as tables and indexes will not be affected although packages and cached dynamic statements are marked invalid. For more information, refer to the "Statement Dependencies" section of the *Administration Guide*.

As in the case of a table hierarchy, it is possible to drop an entire view hierarchy in one statement by naming the root view of the hierarchy, as in the following example:

```
DROP VIEW HIERARCHY VPerson;
```

For more information on dropping and creating views, refer to the *SQL Reference*.

Querying a Typed Table

If you have the required SELECT authority, you can query a typed table in the same way that you query non-typed tables. The query returns the requested columns from the qualifying rows from the target of the SELECT and all of its subtables. For example, the following query on the data in the `Person` table hierarchy returns the names and ages of all people; that is, all rows in the `Person` table and its subtables. For information on writing a similar query if

one of the columns is a structured type column, see "Retrieving and Modifying Structured Type Values" on page 324.

```
SELECT Name, Age
   FROM Person;
```

The result of the query is as follows:

```
NAME                 AGE
-------------------- -----------
Andrew               29
Bob                  30
Cathy                25
Dennis               26
Eva                  31
Franky               28
Gordon               19
Helen                20
Iris                 35
Christina            10
Ken                  55
Leo                  35
Brian                 7
Susan                39
```

Queries that Dereference References

Whenever you have a scoped reference, you can use a *dereference operation* to issue queries that would otherwise require outer joins or correlated subqueries. Consider the `Dept` attribute of the `Employee` table, and subtables of `Employee`, which is scoped to the `BusinessUnit` table. The following example returns the names, salaries, and department names, or NULL values where applicable, of all the employees in the database; that means the query returns these values for every row in the `Employee` table and the `Employee` subtables. You could write a similar query using a correlated subquery or an outer join. However, it is easier to use the *dereference operator* (->) to traverse the path from the reference column in the `Employee` table and subtables to the `BusinessUnit` table, and to return the result from the `Name` column of the `BusinessUnit` table.

The simple format of the dereference operation is as follows:

```
scoped-reference-expression -> column-in-target-typed-table
```

The following query uses the dereference operator to obtain the `Name` column from the `BusinessUnit` table:

```
SELECT Name, Salary, Dept->Name
   FROM Employee
```

The result of the query is as follows:

```
NAME                 SALARY      NAME
-------------------- ----------- --------------------
Dennis               30000       Toy
```

```
Eva                 45000       Shoe
Franky              39000       Shoe
Iris                55000       Toy
Christina           85000       Toy
Ken                 105000      Shoe
Leo                 92000       Shoe
Brian               112000      Toy
Susan               37000.48    ---
```

You can dereference self-referencing references as well. Consider the Parts table defined in Figure 13 on page 310. The following query lists the parts directly used in a wing with the locations of the suppliers of the parts:

```
SELECT P.Descript, P.Supplied_by ->Location
   FROM Parts P
   WHERE P.Used_in -> Descript='Wing';
```

DEREF Built-in Function

You can also dereference references to obtain entire structured objects as a single value by using the DEREF built-in function. The simple form of DEREF is as follows:

```
DEREF (scoped-reference-expression)
```

DEREF is usually used in the context of other built-in functions, such as TYPE_NAME, or to obtain a whole structured object for the purposes of binding out to an application.

Other Type-related Built-in Functions

The DEREF function is often invoked as part of the TYPE_NAME, TYPE_ID, or TYPE_SCHEMA built-in functions. The purpose of these functions, respectively, is to return the name, internal ID, and schema name of the dynamic type of an expression. For example, the following example creates a `Project` typed table with an attribute called `Responsible`:

```
CREATE TYPE Project_t
   AS (Projid INT, Responsible REF(Employee_t))
   MODE DB2SQL;

CREATE TABLE Project
   OF Project_t (REF IS Oid USER GENERATED,
   Responsible WITH OPTIONS SCOPE Employee);
```

The `Responsible` attribute is defined as a reference to the `Employee` table, so that it can refer to instances of managers and architects as well as employees. If your application needs to know the name of the dynamic type of every row, you can use a query like the following:

```
SELECT Projid, Responsible->Name,
   TYPE_NAME(DEREF(Responsible))
   FROM PROJECT;
```

The preceding example uses the dereference operator to return the value of Name from the Employee table, and invokes the DEREF function to return the dynamic type for the instance of Employee_t.

For more information about the built-in functions described in this section, refer to the *SQL Reference*.

Authorization requirement: To use the DEREF function, you must have SELECT authority on every table and subtable in the referenced portion of the table hierarchy. In the above query, for example, you need SELECT privileges on the Employee, Manager, and Architect typed tables.

Additional Query Specification Techniques

Returning Objects of a Particular Type Using ONLY

To have a query return only objects of a particular type, and not of its subtypes, use the ONLY keyword. For example, the following query returns only the names of employees that are not architects or managers:

```
SELECT Name
FROM ONLY(Employee);
```

The previous query returns the following result:

```
NAME
--------------------
Dennis
Eva
Franky
Susan
```

To protect the security of the data, the use of ONLY requires the SELECT privilege on every subtable of Employee.

You can also use the ONLY clause to restrict the operation of an UPDATE or DELETE statement to the named table. That is, the ONLY clause ensures that the operation does not occur on any subtables of that named table.

Restricting Returned Types Using a TYPE Predicate

If you want a more general way to restrict what rows are returned or affected by an SQL statement, you can use the type predicate. The type predicate enables you to compare the dynamic type of an expression to one or more named types. A simple version of the type predicate is:

```
<expression> IS OF (<type_name>[, ...])
```

where *expression* represents an SQL expression that returns an instance of a structured type, and *type_name* represents one or more structured types with which the instance is compared.

For example, the following query returns people who are greater than 35 years old, and who are either managers or architects:

```
SELECT Name
   FROM Employee E
   WHERE E.Age > 35 AND
   DEREF(E.Oid) IS OF (Manager_t, Architect_t);
```

The previous query returns the following result:

```
NAME
--------------------
Ken
```

Returning All Possible Attributes Using OUTER

When DB2 returns a structured type row value, the application does not necessarily know which attributes that particular instance contains or can contain. For example, when you return a person, that person might just have the attributes of a person, or it might have attributes of an employee, manager, or other subtype of person. If your application needs to obtain the values of all possible attributes within one SQL query, you can use the keyword OUTER in the table reference.

OUTER (*table-name*) and OUTER(*view-name*) return a virtual table that consists of the columns of the table or view followed by the additional columns introduced by each of its subtables, if any. The additional columns are added on the right hand side of the table, traversing the subtable hierarchy in the order of depth. Subtables that have a common parent are traversed in the order in which their respective types were created. The rows include all the rows of *table-name* and all of the additional rows of the subtables of *table-name*. Null values are returned for columns that are not in the subtable for the row.

You might use OUTER, for example, when you want to see information about people who tend to achieve above the norm. The following query returns information from the `Person` table hierarchy that have either a high salary `Salary` or a high grade point average `GPA`:

```
SELECT *
   FROM OUTER(Person) P
   WHERE P.Salary > 200000
   OR P.GPA > 3.95 ;
```

Using OUTER(Person) enables you to refer to subtype attributes, which is not otherwise possible in `Person` queries.

The use of OUTER requires the SELECT privilege on every subtable or view of the referenced table because all of their information is exposed through its usage.

Suppose that your application needs to see not just the attributes of these high achievers, but what the most specific type is for each one. You can do this in a single query by passing the object identifier of an object to the TYPE_NAME built-in function and combining it with an OUTER query, as follows:

```
SELECT TYPE_NAME(DEREF(P.Oid)), P.*
   FROM OUTER(Person) P
   WHERE P.Salary > 200000 OR
   P.GPA > 3.95 ;
```

Because the Address column of the Person typed table contains structured types, you would have to define additional functions and issue additional SQL to return the data from that column. For more information on returning data from a structured type column, see "Retrieving and Modifying Structured Type Values" on page 324. Assuming you perform these additional steps, the preceding query returns the following output, where *Additional Attributes* includes GPA and Salary:

```
1                OID          NAME                Additional Attributes
---------------- ------------ ------------------- ...
PERSON_T         a            Andrew              ...
PERSON_T         b            Bob                 ...
PERSON_T         c            Cathy               ...
EMPLOYEE_T       d            Dennis              ...
EMPLOYEE_T       e            Eva                 ...
EMPLOYEE_T       f            Franky              ...
MANAGER_T        i            Iris                ...
ARCHITECT_T      l            Leo                 ...
EMPLOYEE_T       s            Susan               ...
```

Additional Hints and Tips

Defining System-generated Object Identifiers

To have DB2 automatically generate unique object identifiers, you can use the GENERATE_UNIQUE function. Because GENERATE_UNIQUE returns a CHAR (13) FOR BIT DATA value, ensure that your REF USING clause on the CREATE TYPE statement can accommodate a value of that type. The default of VARCHAR (16) FOR BIT DATA is suitable for this purpose. For example, assume that the BusinessUnit_t type is created with the default representation type; that is, no REF USING clause is specified, as follows:

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

The typed table definition is as follows:

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
(REF IS Oid USER GENERATED);
```

Note that you must always provide the clause USER GENERATED.

An INSERT statement to insert a row into the typed table, then, might look like this:

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(GENERATE_UNIQUE( )), 'Toy' 15);
```

To insert an employee that belongs to the Toy department, you can use a statement like the following, which issues a subselect to retrieve the value of the object identifier column from the BusinessUnit table, casts the value to the BusinessUnit_t type, and inserts that value into the Dept column:

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000,
     BusinessUnit_t(SELECT Oid FROM BusinessUnit WHERE Name='Toy'));
```

Creating Constraints on Object Identifier Columns

If you want to use the object identifier column as a key column of the parent table in a foreign key, you must first alter the typed table to add an explicit unique or primary key constraint on the object identifier column. For example, assume that you want to create a self-referencing relationship on employees in which the manager of each employee must always exist as an employee in the employee table, as shown in Figure 14.

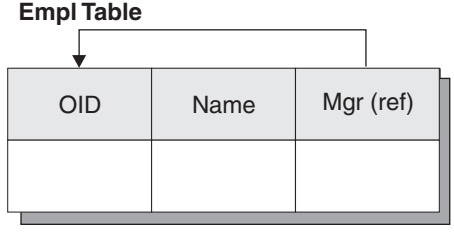


Figure 14. Self-referencing type example

To create a self-referencing relationship, perform the following steps:

Step 1. Create the type

```
CREATE TYPE Empl_t AS
   (Name VARCHAR(10), Mgr REF(Empl_t))
   MODE DB2SQL;
```

Step 2. Create the typed table

```
CREATE TABLE Empl OF Empl_t
   (REF IS Oid USER GENERATED);
```

Step 3. Add the primary or unique constraint on the Oid column:

```
ALTER TABLE Empl ADD CONSTRAINT pk1 UNIQUE(Oid);
```

Step 4. Add the foreign key constraint.

```
ALTER TABLE Empl ADD CONSTRAINT fk1 FOREIGN KEY(Mgr)
   REFERENCES Empl (Oid);
```

Creating and Using Structured Types as Column Types

This section describes the major tasks involved in using a user-defined structured type as the type of a column. Before reading this section, you should be familiar with the material in "Structured Types Overview" on page 292.

Inserting Structured Type Instances into a Column

Structured types can be used in the context of tables, views, or columns. When you create a structured type, you can encapsulate both user-defined type behavior and type attributes. To include behavior for a type, specify a method signature with the CREATE TYPE or ALTER TYPE statement. For more information on creating methods, see "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373.

Figure 15 shows the type hierarchy used as an example in this section. The root type is `Address_t`, which has three subtypes, each with an additional attribute that reflects some aspect of how addresses are formed in that country.

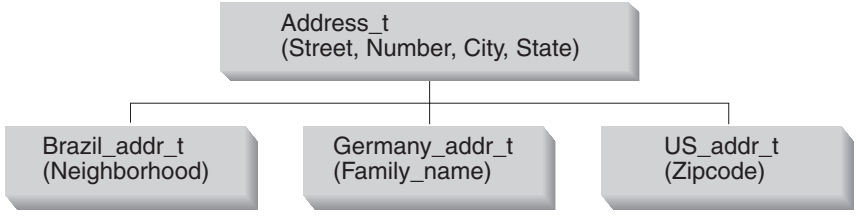


Figure 15. Structured type hierarchy for Address_t type

```
CREATE TYPE Address_t AS
   (street VARCHAR(30),
   number CHAR(15),
   city VARCHAR(30),
   state VARCHAR(10))
   MODE DB2SQL;

CREATE TYPE Germany_addr_t UNDER Address_t AS
   (family_name VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE Brazil_addr_t UNDER Address_t AS
   (neighborhood VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE US_addr_t UNDER Address_t AS
   (zip CHAR(10))
   MODE DB2SQL;
```

Inserting Structured Type Attributes Into Columns

To insert an attribute of a user-defined structured type into a column that is of the same type as the attribute using embedded static SQL, enclose the host variable that represents the instance of the type in parentheses, and append the double-dot operator and attribute name to the closing parenthesis. For example, consider the following situation:

```
- PERSON_T is a structured type that includes the attribute NAME
  of type VARCHAR(30).
- T1 is a table that includes a column C1 of type VARCHAR(30).
- personhv is the host variable declared for type PERSON_T in the
  programming language.
```

The proper syntax for inserting the NAME attribute into column C1 is:

```
EXEC SQL INSERT INTO T1 (C1) VALUES ((:personhv)..NAME)
```

Defining Tables with Structured Type Columns

Unless you are concerned with how structured types are laid out in the data record, there is no additional syntax for creating tables with columns of structured types. For example, the following statement adds a column of `Address_t` type to a `Customer_List` untyped table:

```
ALTER TABLE Customer_List
   ADD COLUMN Address Address_t;
```

Now instances of `Address_t` or any of the subtypes of `Address_t` can be stored in this table. For information on inserting structured types, see "Inserting Rows that Contain Structured Type Values" on page 323.

If you are concerned with how structured types are laid out in the data record, you can use the INLINE LENGTH clause in the CREATE TYPE statement to indicate the maximum size of an instance of a structured type column to store inline with the rest of the values in the row. For more information on the INLINE LENGTH clause, refer to the CREATE TYPE (Structured) statement in the *SQL Reference*.

Defining Types with Structured Type Attributes

A type can be created with a structured type attribute, or it can be altered (before it is used) to add or drop such an attribute. For example, the following CREATE TYPE statement contains an attribute of type `Address_t`:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   REF USING VARCHAR(13)
   MODE DB2SQL;
```

`Person_t` can be used as the type of a table, the type of a column in a regular table, or as an attribute of another structured type.

Inserting Rows that Contain Structured Type Values

When you create a structured type, DB2 automatically generates a constructor method for the type, and generates mutator and observer methods for the attributes of the type. You can use these methods to create instances of structured types, and insert these instances into a column of a table.

Assume that you want to add a new row to the Employee typed table, and that you want that row to contain an address. Just as with built-in data types, you can add this row using INSERT with the VALUES clause. However, when you specify the value to insert into the address, you must invoke the system-provided constructor function to create the value:

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
   US_addr_t ( ) 1
      ..street('Bakely Avenue') 2
      ..number('555') 3
      ..city('San Jose') 4
      ..state('CA') 5
      ..zip('95141')); 6
```

The previous statement creates an instance of the US_addr_t type by performing the following tasks:

1. The call to US_addr_t() invokes the constructor function for the US_addr_t type to create an instance of the type with all attributes set to null values.
2. The call to ..street('Bakely Avenue') invokes the mutator method for the street attribute to set its value to 'Bakely Avenue'.
3. The call to ..number('555') invokes the mutator method for the number attribute to set its value to '555'.
4. The call to ..city('San Jose') invokes the mutator method for the city attribute to set its value to 'San Jose'.
5. The call to ..state('CA') invokes the mutator method for the state attribute to set its value to 'CA'.
6. The call to ..zip('95141') invokes the mutator method for the zip attribute to set its value to '95141'.

Notice that although the type of the column Address in the Employee table is defined with type Address_t, the property of substitutability means that you can can populate it with an instance of US_addr_t because US_addr_t is a subtype of Address_t.

To avoid having to explicitly call the mutator methods for each attribute of a structured type every time you create an instance of the type, consider defining your own SQL-bodied constructor function that initializes all of the attributes. The following example contains the declaration for an SQL-bodied constructor function for the US_addr_t type:

```
CREATE FUNCTION US_addr_t
      (street Varchar(30),
      number Char(15),
      city Varchar(30),
      state Varchar(20),
      zip Char(10))
   RETURNS US_addr_t
   LANGUAGE SQL
   RETURN Address_t()..street(street)..number(number)
       ..city(city)..state(state)..zip(zip);
```

The following example demonstrates how to create an instance of the US_addr_t type by calling the SQL-bodied constructor function from the previous example:

```
INSERT INTO Employee(Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
      US_addr_t('Bakely Avenue', '555', 'San Jose', 'CA', '95141'));
```

Retrieving and Modifying Structured Type Values

There are several ways that applications and user-defined functions can access data in structured type columns. If you want to treat an object as a single value, you must first define *transform functions*, which are described in "Creating the Mapping to the Host Language Program: Transform Functions" on page 329. Once you define the correct transform functions, you can select a structured object much as you can any other value:

```
SELECT Name, Dept, Address
   FROM Employee
   WHERE Salary > 20000;
```

In this section, however, we describe a way of explicitly accessing individual attributes of an object by invoking the DB2 built-in observer and mutator methods. The built-in methods do not require you to define a transform function.

Retrieving Attributes

To explicitly access individual attributes of an object, invoke the DB2 built-in *observer* methods on those attributes. Using the observer methods, you can retrieve the attributes individually rather than treating the object as a single value.

The following example accesses data in the `Address` column by invoking the observer methods on `Address_t`, the defined static type for the `Address` column:

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   FROM Employee
   WHERE Salary > 20000;
```

Note: DB2 enables you to invoke methods that take no parameters using either *<type-name>..<method-name>()* or *<type-name>..<method-name>*, where *type-name* represents the name of the structured type, and *attribute-name* represents the name of the method that takes no parameters.

You can also use observer methods to select each attribute into a host variable, as follows:

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   INTO :name, :dept, :street, :number, :city, :state
   FROM Employee
   WHERE Empno = '000250';
```

Accessing the Attributes of Subtypes

In the `Employee` table, addresses can be of 4 different types: `Address_t`, `US_addr_t`, `Brazil_addr_t`, and `Germany_addr_t`. The previous example accesses only the attributes of the static type `Address_t`. To access attributes of values from one of the subtypes of `Address_t`, you *must* use the TREAT expression to indicate to DB2 that a particular object can be of the `US_addr_t`, `Germany_addr_t`, or `Brazil_addr_t` types. The TREAT expression casts a structured type expression into one of its subtypes, as shown in the following query:

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state,
   CASE
      WHEN Address IS OF (US_addr_t)
      THEN TREAT(Address AS US_addr_t)..zip
      WHEN Address IS OF (Germany_addr_t)
      THEN TREAT (Address AS Germany_addr_t)..family_name
      WHEN Address IS OF (Brazil_addr_t)
      THEN TREAT (Address AS Brazil_addr_t)..neighborhood
   ELSE NULL END
   FROM Employee
   WHERE Salary > 20000;
```

Note: You can only use the preceding approach to determine the subtype of a structured type when the attributes of the subtype are all of the same type, or can be cast to the same type. In the previous example, `zip`, `family_name`, and `neighborhood` are all VARCHAR or CHAR types, and can be cast to the same type.

For more information about the syntax of the TREAT expression or the TYPE predicate, refer to the *SQL Reference*.

Modifying Attributes

To change an attribute of a structured column value, invoke the mutator method for the attribute you want to change. For example, to change the

street attribute of an address, you can invoke the mutator method for street with the value to which it will be changed. The returned value is an address with the new value for street. The following example invokes a mutator method for the attribute named street to update an address type in the Employee table:

```
UPDATE Employee
   SET Address = Address..street('Bailey')
   WHERE Address..street = 'Bakely';
```

The following example performs the same update as the previous example, but instead of naming the structured column for the update, the SET clause directly accesses the mutator method for the attribute named street:

```
UPDATE Employee
   SET Address..street = 'Bailey'
   WHERE Address..street = 'Bakely';
```

Returning Information About the Type

As described in "Other Type-related Built-in Functions" on page 316, you can use built-in functions to return the name, schema, or internal type ID of a particular type. The following statement returns the exact type of the address value associated with the employee named 'Iris':

```
SELECT TYPE_NAME(Address)
   FROM Employee
   WHERE Name='Iris';
```

Associating Transforms with a Type

Transform functions naturally occur in pairs: one FROM SQL transform function, and one TO SQL transform function. The FROM SQL function converts a structured type object into a type that can be exchanged with an external program, and the TO SQL function constructs the object. When you create transform functions, you put each logical pair of transform functions into a group. The *transform group* name uniquely identifies a pair of these functions for a given structured type.

Before you can use a transform function, you must use the CREATE TRANSFORM statement to associate the transform function with a group name and a type. The CREATE TRANSFORM statement identifies one or more existing functions and causes them to be used as transform functions. The following example names two pairs of functions to be used as transform functions for the type Address_t. The statement creates two transform groups, func_group and client_group, each of which consists of a FROM SQL transform and a TO SQL transform.

```
CREATE TRANSFORM FOR Address_t
   func_group ( FROM SQL WITH FUNCTION addresstofunc,
      TO SQL WITH FUNCTION functoaddress )
   client_group ( FROM SQL WITH FUNCTION stream_to_client,
      TO SQL WITH FUNCTION stream_from_client ) ;
```

You can associate additional functions with the Address_t type by adding more groups on the CREATE TRANSFORM statement. To alter the transform definition, you must reissue the CREATE TRANSFORM statement with the additional functions. For example, you might want to customize your client functions for different host language programs, such as having one for C and one for Java. To optimize the performance of your application, you might want your transforms to work only with a subset of the object attributes. Or you might want one transform that uses VARCHAR as the client representation for an object and one transform that uses BLOB.

Use the SQL statement DROP TRANSFORM to disassociate transform functions from types. After you execute the DROP TRANSFORM statement, the functions will still exist, but they will no longer be used as transform functions for this type. The following example disassociates the specific group of transform functions func_group for the Address_t type, and then disassociates all transform functions for the Address_t type:

```
DROP TRANSFORMS func_group FOR Address_t;

DROP TRANSFORMS ALL FOR Address_t;
```

Recommendations for Naming Transform Groups

Transform group names are *unqualified* identifiers; that is, they are not associated with any specific schema. Unless you are writing transforms to handle subtype parameters, as described in "Retrieving Subtype Data from DB2 (Bind Out)" on page 341, you should not assign a different transform group name for every structured type. Because you might need to use several different, unrelated types in the same program or in the same SQL statement, you should name your transform groups according to the tasks performed by the transform functions.

The names of your transform groups should generally reflect the function they perform without relying on type names or in any way reflecting the logic of the transform functions, which will likely be very different across the different types. For example, you could use the name func_group or object_functions for any group in which your TO and FROM SQL function transforms are defined. You could use the name client_group or program_group for a group that contains TO and FROM SQL client transforms.

In the following example, the Address_t and Polygon types use very different transforms, but they use the same function group names

```
CREATE TRANSFORM FOR Address_t
   func_group (TO SQL WITH FUNCTION functoaddress,
   FROM SQL WITH FUNCTION addresstofunc );

CREATE TRANSFORM FOR Polygon
   func_group (TO SQL WITH FUNCTION functopolygon,
   FROM SQL WITH FUNCTION polygontofunc);
```

Once you set the transform group to `func_group` in the appropriate situation, as described in "Where Transform Groups Must Be Specified", DB2 invokes the correct transform function whenever you bind in or bind out an address or polygon.

Restriction: Do not begin a transform group with the string 'SYS'; this group is reserved for use by DB2.

When you define an external function or method and you do not specify a transform group name, DB2 attempts to use the name DB2_FUNCTION, and assumes that that group name was specified for the given structured type. If you do not specify a group name when you precompile a client program that references a given structured type, DB2 attempts to use a group name called DB2_PROGRAM, and again assumes that the group name was defined for that type.

This default behavior is convenient in some cases, but in a more complex database schema, you might want a slightly more extensive convention for transform group names. For example, it may help you to use different group names for different languages to which you might bind out the type.

Where Transform Groups Must Be Specified

Considering that there can be many transform groups defined for a given structured type, you must specify which group of transforms to use for that type in a program or specific SQL statement. There are three circumstances in which you must specify transform groups:

- When an external function or method is defined, you must specify the group that *decomposes* and *constructs* a referenced object. For more information, see "Specifying Transform Groups for External Routines".
- When precompiling or binding static SQL, you must specify the group of transforms that perform client bind in and bind out for a referenced type. For more information, see "Setting the Transform Group for Static SQL" on page 329.
- When executing dynamic SQL, or when using the command line processor, you must specify the group of transforms which perform client bind in and bind out for a referenced type. For more information, see "Setting the Transform Group for Dynamic SQL" on page 329.

Specifying Transform Groups for External Routines

The CREATE FUNCTION and CREATE METHOD statements enable you to specify the TRANSFORM GROUP clause, which is only valid when the value of the LANGUAGE clause is not SQL. SQL language functions do not require transforms, while external functions do require transforms. The TRANSFORM GROUP clause allows you to specify, for any given function or method, the transform group that contains the TO SQL and FROM SQL transforms used for structured type parameters and results. In the following example, the

CREATE FUNCTION and CREATE METHOD statements specify the transform group func_group for the TO SQL and FROM SQL transforms:

```
CREATE FUNCTION stream_from_client (VARCHAR (150))
   RETURNS Address_t
   ...
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_stream_from_client'
   ...

CREATE METHOD distance ( point )
   FOR polygon
   RETURNS integer
   :
   TRANSFORM GROUP func_group ;
```

Setting the Transform Group for Dynamic SQL

If you use dynamic SQL, you can set the CURRENT DEFAULT TRANSFORM GROUP special register. This special register is not used for static SQL statements or for the exchange of parameters and results with external functions or methods. Use the SET CURRENT DEFAULT TRANSFORM GROUP statement to set the default transform group for your dynamic SQL statements:

```
SET CURRENT DEFAULT TRANSFORM GROUP = client_group;
```

Setting the Transform Group for Static SQL

For static SQL, use the TRANSFORM GROUP option on the PRECOMPILE or BIND command to specify the static transform group used by static SQL statements to exchange values of various types with host programs. Static transform groups do not apply to dynamic SQL statements, or to the exchange of parameters and results with external functions or methods. To specify the static transform group on the PRECOMPILE or BIND command, use the TRANSFORM GROUP clause:

```
PRECOMPILE ...
TRANSFORM GROUP client_group
... ;
```

For more information on the PRECOMPILE and BIND commands, refer to the *Command Reference*.

Creating the Mapping to the Host Language Program: Transform Functions

An application cannot directly select an entire object, although, as described in "Retrieving Attributes" on page 324, you can select individual attributes of an object into an application. An application usually does not directly insert an entire object, although it can insert the result of an invocation of the constructor function:

```
INSERT INTO Employee(Address) VALUES (Address_t());
```

To exchange whole objects between the server and client applications, or external functions, you must normally write *transform functions*.

A transform function defines how DB2 converts an object into a well-defined format for accessing its contents, or *binds out* the object. A different transform function defines how DB2 returns the object to be stored in the database, or *binds in* the object. Transforms that bind out an object are called FROM SQL transform functions, and transforms that bind in a column object are called TO SQL transforms.

Most likely, there will be different transforms for passing objects to *routines*, or external UDFs and methods, than those for passing objects to client applications. This is because when you pass the object to an external routine, you *decompose* the object and pass it to the routine as a list of parameters. With client applications, you must turn the object into a single built-in type, such as a BLOB. This process is called encoding the object. Often these two types of transforms are used together.

Use the SQL statement CREATE TRANSFORM to associate transform functions with a particular structured type. Within the CREATE TRANSFORM statement, the functions are paired into what are called *transform groups*. This makes it easier to identify which functions are used for a particular transform purpose. Each transform group can contain not more than one FROM SQL transform, and not more than one TO SQL transform, for a particular type.

Note: The following topics cover the simple case in which the application always receives a known exact type, such as `Address_t`. These topics do not describe the likely scenario in which an external routine or a client program may receive `Address_t`, `Brazil_addr_t`, `Germany_addr_t`, or `US_addr_t`. However, you must understand the basic process before attempting to apply that basic process to the more complex case, in which the external routine or client needs to handle dynamically any type or its subtypes. For information about how to dynamically handle subtype instances, see "Retrieving Subtype Data from DB2 (Bind Out)" on page 341.

Exchanging Objects with External Routines: Function Transforms

This section describes a particular type of transforms called *function transforms*. DB2 uses these TO SQL and FROM SQL function transforms to pass an object to and from an external routine. There is no need to use transforms for SQL-bodied routines. However, as "Exchanging Objects with a Program: Client Transforms" on page 335 describes, DB2 often uses these functions as part of the process of passing an object to and from a client program.

The following example issues an SQL statement that invokes an external UDF called MYUDF that takes an address as an input parameter, modifies the address (to reflect a change in street names, for example), and returns the modified address:

```
SELECT MYUDF(Address)
FROM PERSON;
```

Figure 16 shows how DB2 processes the address.

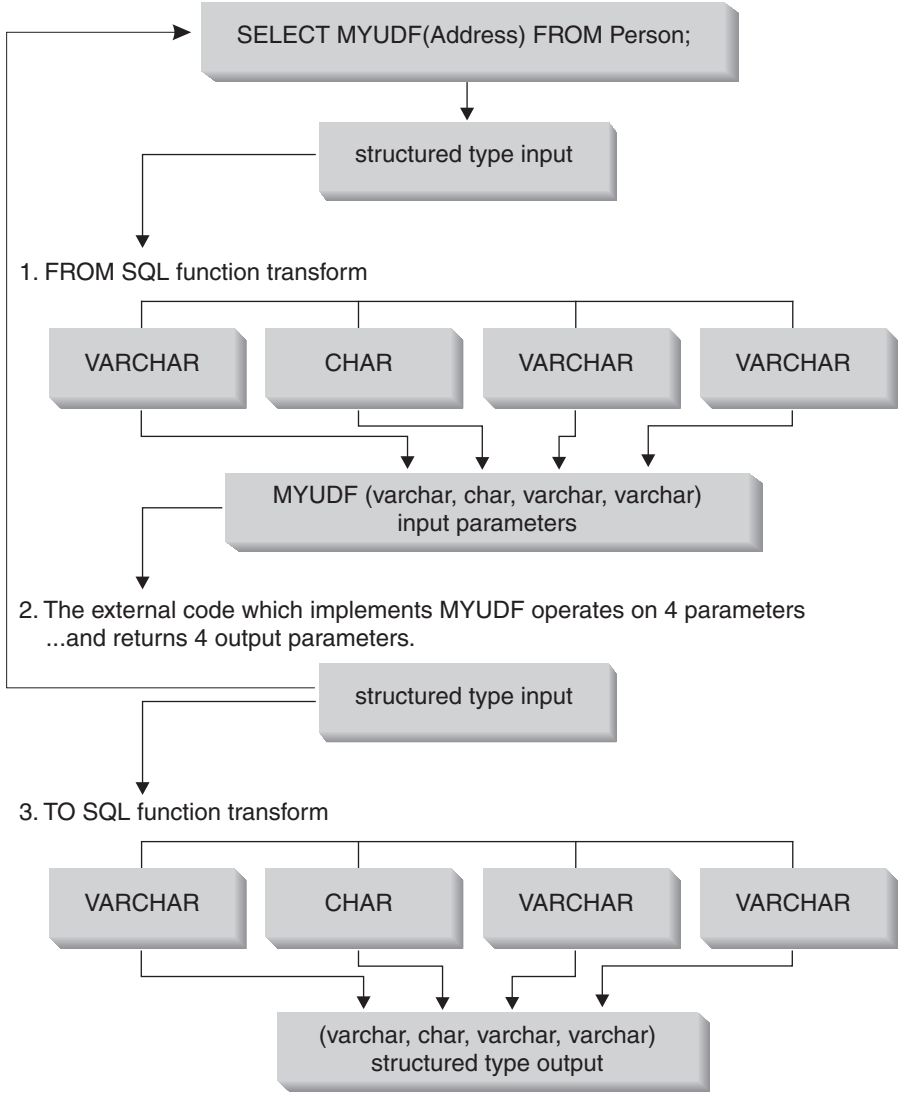


Figure 16. Exchanging a structured type parameter with an external routine

1. Your FROM SQL transform function decomposes the structured object into an ordered set of its base attributes. This enables the routine to receive the object as a simple list of parameters whose types are basic built-in data types. For example, assume that you want to pass an address object to an external routine. The attributes of `Address_t` are VARCHAR, CHAR, VARCHAR, and VARCHAR, in that order. The FROM SQL transform for passing this object to a routine must accept this object as an input and return VARCHAR, CHAR, VARCHAR, and VARCHAR. These outputs are then passed to the external routine as four separate parameters, with four corresponding null indicator parameters, and a null indicator for the structured type itself. The order of parameters in the FROM SQL function does not matter, as long as all functions that return `Address_t` types use the same order. For more information, see "Passing Structured Type Parameters to External Routines" on page 334.
2. Your external routine accepts the decomposed address as its input parameters, does its processing on those values, and then returns the attributes as output parameters.
3. Your TO SQL transform function must turn the VARCHAR, CHAR, VARCHAR, and VARCHAR parameters that are returned from `MYUDF` back into an object of type `Address_t`. In other words, the TO SQL transform function must take the four parameters, and all of the corresponding null indicator parameters, as output values from the routine. The TO SQL function constructs the structured object and then mutates the attributes with the given values.

Note: If `MYUDF` also returns a structured type, another transform function must transform the resultant structured type when the UDF is used in a SELECT clause. To avoid creating another transform function, you can use SELECT statements with observer methods, as in the following example:

```
SELECT Name
   FROM Employee
   WHERE MYUDF(Address)..city LIKE 'Tor%';
```

Implementing Function Transforms Using SQL-Bodied Routines: To decompose and construct objects when exchanging the object with an external routine, you can use user-defined functions written in SQL, called SQL-bodied routines. To create a SQL-bodied routine, issue a CREATE FUNCTION statement with the LANGUAGE SQL clause.

In your SQL-bodied function, you can use constructors, observers, and mutators to achieve the transformation. As shown in Figure 16 on page 331, this SQL-bodied transform intervenes between the SQL statement and the external function. The FROM SQL transform takes the object as an SQL parameter and returns a row of values representing the attributes of the

structured type. The following example contains a possible FROM SQL transform function for an address object using a SQL-bodied function:

```
CREATE FUNCTION addresstofunc (A Address_t) 1
   RETURNS ROW  (Street VARCHAR(30), Number CHAR(15),
      City VARCHAR(30), State (VARCHAR(10)) 2

   LANGUAGE SQL 3
   RETURN VALUES (A..Street, A..Number, A..City, A..State) 4
```

The following list explains the syntax of the preceding CREATE FUNCTION statement:

1. The signature of this function indicates that it accepts one parameter, an object of type `Address_t`.
2. The RETURNS ROW clause indicates that the function returns a row containing four columns: `Street`, `Number`, `City`, and `State`.
3. The LANGUAGE SQL clause indicates that this is an SQL-bodied function, not an external function.
4. The RETURN clause marks the the beginning of the function body. The body consists of a single VALUES clause that invokes the observer method for each attribute of the `Address_t` object. The observer methods decompose the object into a set of base types, which the function returns as a row.

DB2 does not know that you intend to use this function as a transform function. Until you create a transform group that uses this function, and then specify that transform group in the appropriate situation, DB2 cannot use the function as a transform function. For more information, see "Associating Transforms with a Type" on page 326.

The TO SQL transform simply does the opposite of the FROM SQL function. It takes as input the list of parameters from a routine and returns an instance of the structured type. To construct the object, the following FROM SQL function invokes the constructor function for the `Address_t` type:

```
CREATE FUNCTION functoaddress (street VARCHAR(30), number CHAR(15),
                               city VARCHAR(30), state VARCHAR(10)) 1
   RETURNS Address_t 2
   LANGUAGE SQL
   CONTAINS SQL
   RETURN
      Address_t()..street(street)..number(number)
      ..city(city)..state(state) 3
```

The following list explains the syntax of the previous statement:

1. The function takes a set of base type attributes.
2. The function returns an `Address_t` structured type.

3. The function constructs the object from the input types by invoking the constructor for `Address_t` and the mutators for each of the attributes.

The order of parameters in the FROM SQL function does not matter, other than that all functions that return addresses must use this same order.

Passing Structured Type Parameters to External Routines: When you pass structured type parameters to an external routine, you should pass a parameter for each attribute. You must pass a null indicator for each parameter and a null indicator for the structured type itself. The following example accepts the structured type `Address_t` and returns a base type:

```
CREATE FUNCTION stream_to_client (Address_t)
   RETURNS VARCHAR(150) ...
```

The external routine must accept the null indicator for the instance of the `Address_t` type (address_ind) and one null indicator for each of the attributes of the `Address_t` type. There is also a null indicator for the VARCHAR output parameter. The following code represents the C language function headers for the functions which implement the UDFs:

```
void SQL_API_FN stream_to_client(
/*decomposed address*/
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *output,
/*null indicators for type attributes*/
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
/*null indicator for instance of the type*/
   SQLUDF_NULLIND *address_ind,
/*null indicator for the VARCHAR output*/
   SQLUDF_NULLIND *out_ind,
   SQLUDF_TRAIL_ARGS)
```

Passing Structured Type Parameters to External Routines: Complex: Suppose that the routine accepts two different structured type parameters, *st1* and *st2*, and returns another structured type of *st3*:

```
CREATE FUNCTION myudf (int, st1, st2)
   RETURNS st3
```

Table 14. Attributes of myudf parameters

ST1	ST2	ST3
st1_att1 VARCHAR	st2_att1 VARCHAR	st3_att1 INTEGER
st2_att2 INTEGER	st2_att2 CHAR	st3_att2 CLOB

Table 14. Attributes of myudf parameters (continued)

ST1	ST2	ST3
	st2_att3 INTEGER	

The following code represents the C language headers for routines which implement the UDFs. The arguments include variables and null indicators for the attributes of the decomposed structured type and a null indicator for each instance of a structured type, as follows:

```
void SQL_API_FN myudf(
    SQLUDF_INTEGER *INT,
 /* Decompose st1 input */
    SQLUDF_VARCHAR *st1_att1,
    SQLUDF_INTEGER  *st1_att2,
 /* Decompose st2 input */
    SQLUDF_VARCHAR *st2_att1,
    SQLUDF_CHAR   *st2_att2,
    SQLUDF_INTEGER *st2_att3,
 /* Decompose st3 output */
    SQLUDF_VARCHAR *st3_att1out,
    SQLUDF_CLOB   *st3_att2out,
 /* Null indicator of integer*/
    SQLUDF_NULLIND *INT_ind,
 /* Null indicators of st1 attributes and type*/
    SQLUDF_NULLIND *st1_att1_ind,
    SQLUDF_NULLIND *st1_att2_ind,
    SQLUDF_NULLIND *st1_ind,
 /* Null indicators of st2 attributes and type*/
    SQLUDF_NULLIND *st2_att1_ind,
    SQLUDF_NULLIND *st2_att2_ind,
    SQLUDF_NULLIND *st2_att3_ind,
    SQLUDF_NULLIND *st2_ind,
 /* Null indicators of st3_out attributes and type*/
    SQLUDF_NULLIND *st3_att1_ind,
    SQLUDF_NULLIND *st3_att2_ind,
    SQLUDF_NULLIND *st3_ind,
 /* trailing arguments */
    SQLUDF_TRAIL_ARGS
 )
```

Exchanging Objects with a Program: Client Transforms: This section describes *client* transforms. Client transforms exchange structured types with client application programs.

For example, assume that you want to execute the following SQL statement:

```
...
SQL TYPE IS Address_t AS VARCHAR(150) addhv;
...

EXEC SQL SELECT Address
```

```
    FROM Person
    INTO :addhv
    WHERE AGE > 25
END EXEC;
```

Figure 17 shows the process of binding out that address to the client program.

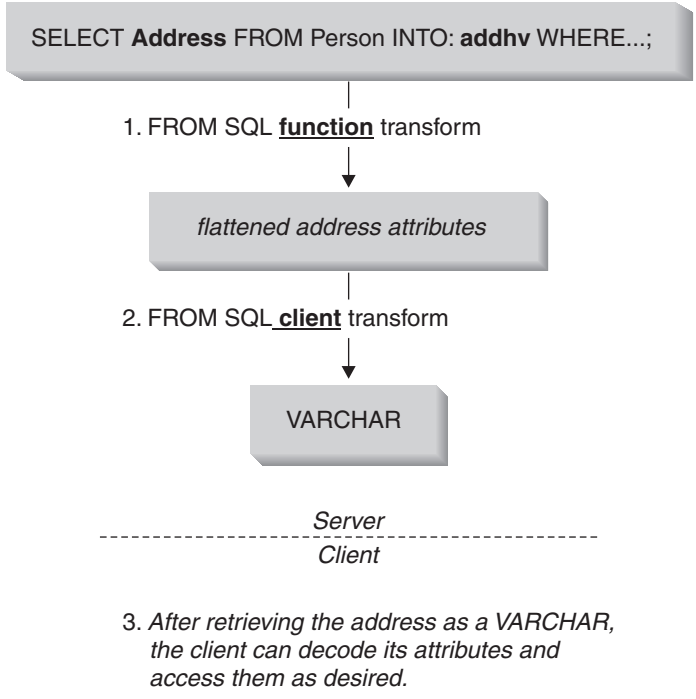


Figure 17. Binding out a structured type to a client application

1. The object must first be passed to the FROM SQL function transform to decompose it into its base type attributes.
2. Your FROM SQL client transform must encode the value into a single built-in type, such as a VARCHAR or BLOB. This enables the client program to receive the entire value in a single host variable.

   This encoding can be as simple as copying the attributes into a contiguous area of storage (providing for required alignments as necessary). Because the encoding and decoding of attributes cannot generally be achieved with SQL, client transforms are usually written as external UDFs.

   For information about processing data between platforms, see "Data Conversion Considerations" on page 339.
3. The client program processes the value.

Figure 18 shows the reverse process of passing the address back to the database.

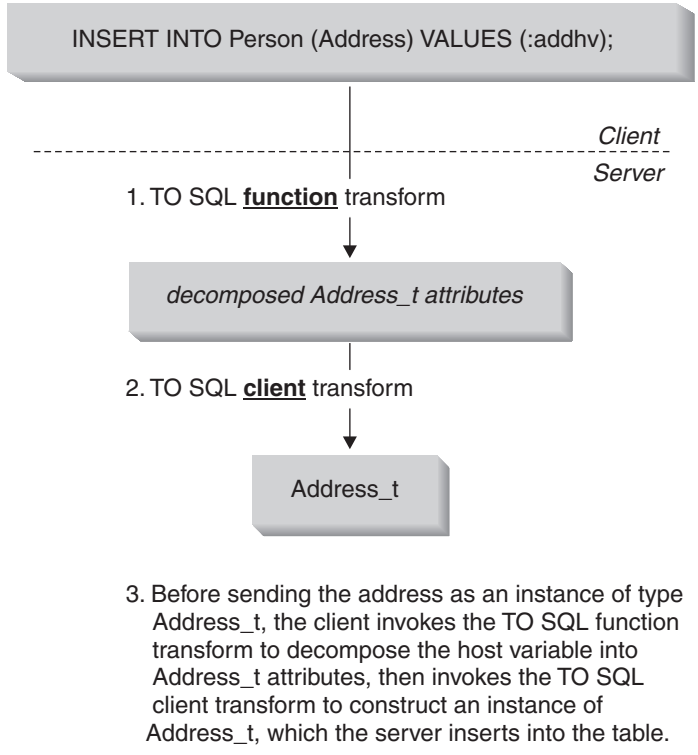


Figure 18. Binding in a structured type from a client

1. The client application encodes the address into a format expected by the TO SQL client transform.
2. The TO SQL client transform decomposes the single built-in type into a set of its base type attributes, which is used as input to the TO SQL function transform.
3. The TO SQL function transform constructs the address and returns it to the database.

Implementing Client Transforms Using External UDFs: Register the client transforms the same way as any other external UDF. For example, assume that you have written external UDFs that do the appropriate encoding and decoding for an address. Suppose that you have named the FROM SQL client transform `from_sql_to_client` and the TO SQL client transform `to_sql_from_client`. In both of these cases, the output of the functions are in a format that can be used as input by the appropriate FROM SQL and TO SQL function transforms.

```
CREATE FUNCTION from_sql_to_client (Address_t)
   RETURNS VARCHAR (150)
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_from_sql_to_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE DB2SQL;
```

The DDL in the previous example makes it seem as if the `from_sql_to_client` UDF accepts a parameter of type `Address_t`. What really happens is that, for each row for which the `from_sql_to_client` UDF is invoked, the Addresstofunc transform decomposes the `Address` into its various attributes. The `from_sql_to_client` UDF produces a simple character string and formats the address attributes for display, allowing you to use the following simple SQL query to display the `Name` and `Address` attributes for each row of the `Person` table:

```
SELECT Name, from_sql_to_client (Address)
FROM Person;
```

Client Transform for Binding in from a Client: The following DDL registers a function that takes the VARCHAR-encoded object from the client, decomposes it into its various base type attributes, and passes it to the TO SQL function transform.

```
CREATE FUNCTION to_sql_from_client (VARCHAR (150))
   RETURNS Address_t
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_to_sql_from_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE DB2SQL;
```

Although it appears as if the `to_sql_from_client` returns the address directly, what really happens is that `to_sql_from_client` converts the VARCHAR (150) to a set of base type attributes. Then DB2 implicitly invokes the TO SQL transform `functoaddress` to construct the address object that is returned to the database.

How does DB2 know which function transform to invoke? Notice that the DDL in both `to_sql_from_client` and `from_sql_to_client` include a clause called TRANSFORM GROUP. This clause tells DB2 which set of transforms to use in processing the address type in those functions. For more information, see "Associating Transforms with a Type" on page 326.

Data Conversion Considerations: It is important to note that when data, especially binary data, is exchanged between server and client, there are several data conversion issues to consider. For example, when data is transferred between platforms with different byte-ordering schemes, numeric data must undergo a byte-reversal process to restore its correct numeric value. Different operating systems also have certain alignment requirements for referencing numeric data in memory; some operating systems will cause program exceptions if these requirements are not observed. Character data types are automatically converted by the database, except when character data is embedded in a binary data type such as BLOB or a VARCHAR FOR BIT DATA.

There are two ways to avoid data conversion problems:

- Always transform objects into printable character data types, including numeric data.

  This approach has the disadvantages of slowing performance, due to the many potential conversions required, and increasing the complexity of code accessing these objects, such as on the client or in the transform function itself.
- Devise a platform-neutral format for an object transformed into a binary data type, similar to the approach that is taken by Java implementations. Be sure to:
  - Take care when packing or unpacking these compacted objects to properly encode or decode the individual data types and to avoid data corruption or program faults.
  - Include sufficient header information in the transformed type so that the remainder of the encoded object can be correctly interpreted independent of the client or server platform.
  - Use the DBINFO option of CREATE FUNCTION to pass to the transform function various characteristics related to the database server environment. These characteristics can be included in the header in a platform-neutral format. For more information about using DBINFO, see "The Arguments Passed from DB2 to a UDF" on page 395.

  For more information about data conversion, see "National Language Support Considerations" on page 503.

Note: As much as possible, you should write transform functions so that they correctly handle all of the complexities associated with the transfer of data between server and client. When you design your application, consider the specific requirements of your environment and evaluate the tradeoffs between complete generality and simplicity. For example, if you know that both the database server and all of its clients run in an AIX environment and use the same code page, you could decide to ignore the previously discussed considerations, because no conversions

are currently required. However, if your environment changes in the future, you may have to exert considerable effort to revise your original design to correctly handle data conversion.

Transform Function Summary

Table 15 is intended to help you determine what transform functions you need, depending on whether you are binding out to an external routine or a client application.

Table 15. Characteristics of transform functions

Characteristic	**Exchanging values with an external routine**		**Exchanging values with a client application**	
Transform direction	FROM SQL	TO SQL	FROM SQL	TO SQL
What is being transformed	Routine parameter	Routine result	Output host variable	Input host variable
Behavior	Decomposes	Constructs	Encodes	Decodes
Transform function parameters	Structured type	Row of built-in types	Structured type	One built-in type
Transform function result	Row of built-in types (probably attributes)	Structured type	One built-in type	Structured type
Dependent on another transform?	No	No	FROM SQL UDF transform	TO SQL UDF transform
When is the transform group specified?	At the time the UDF is registered		Static: precompile time Dynamic: Special register	
Are there data conversion considerations?	No		Yes	

Note: Although not generally the case, client type transforms can actually be written in SQL if any of the following are true:

- The structured type contains only one attribute.
- The encoding and decoding of the attributes into a built-in type can be achieved by some combination of SQL operators or functions.

In these cases, you do not have to depend on function transforms to exchange the values of a structured type with a client application.

Retrieving Subtype Data from DB2 (Bind Out)

Most of the information in the previous sections assume that the application is passing around a known exact type. If your data model takes advantage of subtypes, a value in a column could be one of many different subtypes. This section describes how you can dynamically choose the correct transform functions based on the actual input type.

Suppose you want to issue the following SELECT statement:

```
SELECT Address
   FROM Person
   INTO :hvaddr;
```

The application has no way of knowing whether a instance of `Address_t`, `US_addr_t`, or so on, will be returned. To keep the example from being too complex, let us assume that only `Address_t` or `US_addr_t` can be returned. The structures of these types are different, so the transforms that decompose the attributes must be different. To ensure that the proper transforms are invoked, perform the following steps:

Step 1. Create a FROM SQL function transform for each variation of address:

```
CREATE FUNCTION addresstofunc(A address_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10))
   LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State)

CREATE FUNCTION US_addresstofunc(A US_addr_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10), Zip
   CHAR(10))
   LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State, A..Zip)
```

Step 2. Create transform groups, one for each type variation:

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (FROM SQL WITH FUNCTION addresstofunc)

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (FROM SQL WITH FUNCTION US_addresstofunc)
```

Step 3. Create external UDFs, one for each type variation.

Register the external UDF for the `Address_t` type:

```
CREATE FUNCTION address_to_client (A Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!address_to_client'
   ...
   TRANSFORM GROUP funcgroup1
```

Write the address_to_client UDF:

```
void SQL_API_FN address_to_client(
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *output,

   /* Null indicators for attributes */
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   /* Null indicator for instance */
   SQLUDF_NULLIND *address_ind,
   /* Null indicator for output */
   SQLUDF_NULLIND *output_ind,
   SQLUDF_TRAIL_ARGS)

{
   sprintf (output, "[address_t] [Street:%s] [number:%s]
   [city:%s] [state:%s]",
   street, number, city, state);
   *output_ind = 0;
}
```

Register the external UDF for the US_addr_t type:

```
CREATE FUNCTION address_to_client (A US_addr_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!US_addr_to_client'
   ...
   TRANSFORM GROUP funcgroup2
```

Write the US_addr_to_client UDF:

```
void SQL_API_FN US_address_to_client(
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,
   SQLUDF_CHAR     *zip,
   SQLUDF_VARCHAR  *output,

   /* Null indicators */
   SQLUDF_NULLIND  *street_ind,
   SQLUDF_NULLIND  *number_ind,
   SQLUDF_NULLIND  *city_ind,
   SQLUDF_NULLIND  *state_ind,
   SQLUDF_NULLIND  *zip_ind,
   SQLUDF_NULLIND  *us_address_ind,
   SQLUDF_NULLIND  *output_ind,
   SQLUDF_TRAIL_ARGS)
```

```
{
   sprintf (output, "[US_addr_t] [Street:%s] [number:%s]
   [city:%s] [state:%s] [zip:%s]",
   street, number, city, state, zip);
   *output_ind = 0;
}
```

Step 4. Create a SQL-bodied UDF that chooses the correct external UDF to process the instance. The following UDF uses the TREAT specification in SELECT statements combined by a UNION ALL clause to invoke the correct FROM SQL client transform:

```
CREATE FUNCTION addr_stream (ab Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE SQL
   RETURN
   WITH temp(addr) AS
   (SELECT address_to_client(ta.a)
      FROM TABLE (VALUES (ab)) AS ta(a)
      WHERE ta.a IS OF (ONLY Address_t)
      UNION ALL
   SELECT address_to_client(TREAT (tb.a AS US_addr_t))
      FROM TABLE (VALUES (ab)) AS tb(a)
      WHERE tb.a IS OF (ONLY US_addr_t))
   SELECT addr FROM temp;
```

At this point, applications can invoke the appropriate external UDF by invoking the `Addr_stream` function:

```
SELECT Addr_stream(Address)
   FROM Employee;
```

Step 5. Add the `Addr_stream` external UDF as a FROM SQL *client* transform for `Address_t`:

```
CREATE TRANSFORM GROUP FOR Address_t
   client_group (FROM SQL
   WITH FUNCTION Addr_stream)
```

Note: If your application might use a type predicate to specify particular address types in the query, add `Addr_stream` as a FROM SQL to client transform for `US_addr_t`. This ensures that `Addr_stream` can be invoked when a query specifically requests instances of `US_addr_t`.

Step 6. Bind the application with the TRANSFORM GROUP option set to `client_group`.

```
PREP myprogram TRANSFORM GROUP client_group
```

When DB2 binds the application that contains the `SELECT Address FROM Person INTO :hvar` statement, DB2 looks for a FROM SQL client transform. DB2 recognizes that a structured type is being bound out, and looks in the transform group `client_group` because that is the TRANSFORM GROUP specified at bind time in 6.

The transform group contains the transform function Addr_stream associated with the root type Address_t in 5 on page 343. Addr_stream is a SQL-bodied function, defined in 4 on page 343, so it has no dependency on any other transform function. The Addr_stream function returns VARCHAR(150), the data type required by the :hvaddr host variable.

The Addr_stream function takes an input value of type Address_t, which can be substituted with US_addr_t in this example, and determines the dynamic type of the input value. When Addr_stream determines the dynamic type, it invokes the corresponding external UDF on the value: address_to_client if the dynamic type is Address_t; or USaddr_to_client if the dynamic type is US_addr_t. These two UDFs are defined in 3 on page 341. Each UDF decomposes their respective structured type to VARCHAR(150), the type required by the Addr_stream transform function.

To accept the structured types as input, each UDF needs a FROM SQL transform function to decompose the input structured type instance into individual attribute parameters. The CREATE FUNCTION statements in 3 on page 341 name the TRANSFORM GROUP that contains these transforms.

The CREATE FUNCTION statements for the transform functions are issued in 1 on page 341. The CREATE TRANSFORM statements that associate the transform functions with their transform groups are issued in 2 on page 341.

Returning Subtype Data to DB2 (Bind In)

Once the application described in "Retrieving Subtype Data from DB2 (Bind Out)" on page 341 manipulates the address value, it may need to insert the changed value back into the database. Suppose you want to insert a structured type into a DB2 database from an application using the following syntax:

```
INSERT INTO person (Oid, Name, Address)
   VALUES ('n', 'Norm', :hvaddr);
```

To execute the INSERT statement for a structured type, your application must perform the following steps:

Step 1. Create a TO SQL function transform for each variation of address. The following example shows SQL-bodied UDFs that transform the Address_t and US_addr_t types:

```
CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10))
RETURNS Address_t
LANGUAGE SQL
RETURN Address_t()..street(str)..number(num)..city(cy)..state(st);

CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10),
  zp CHAR(10))
```

```
RETURNS US_addr_t
LANGUAGE SQL
RETURN US_addr_t()..street(str)..number(num)..city(cy)
    ..state(st)..zip(zp);
```

Step 2. Create transform groups, one for each type variation:

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (TO SQL
   WITH FUNCTION functoaddress);

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (TO SQL
   WITH FUNCTION functousaddr);
```

Step 3. Create external UDFs that return the encoded address types, one for each type variation.

Register the external UDF for the `Address_t` type:

```
CREATE FUNCTION client_to_address (encoding VARCHAR(150))
   RETURNS Address_t
   LANGUAGE C
   TRANSFORM GROUP funcgroup1
   ...
   EXTERNAL NAME 'address!client_to_address';
```

Write the external UDF for the `Address_t` version of `client_to_address`:

```
void SQL_API_FN client_to_address (
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   SQLUDF_NULLIND *address_ind,
   SQLUDF_TRAIL_ARGS )
{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (number, pc);
```

```
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (city, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (state, pc);

   *street_ind = *number_ind = *city_ind
   = *state_ind = *address_ind = 0;
}
```

Register the external UDF for the US_addr_t type:

```
CREATE FUNCTION client_to_us_address (encoding VARCHAR(150))
   RETURNS US_addr_t
   LANGUAGE C
   TRANSFORM GROUP funcgroup1
   ...
   EXTERNAL NAME 'address!client_to_US_addr';
```

Write the external UDF for the US_addr_t version of client_to_address:

```
void SQL_API_FN client_to_US_addr(
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *zip,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   SQLUDF_NULLIND *zip_ind,
   SQLUDF_NULLIND *us_addr_ind,
   SQLUDF_TRAIL_ARGS)

{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strncpy (number, pc,14);
   pc = strtok (NULL, ":]");
```

```
    pc = strtok (NULL, ":]");
    strcpy (city, pc);
    pc = strtok (NULL, ":]");
    pc = strtok (NULL, ":]");
    strcpy (state, pc);
    pc = strtok (NULL, ":]");
    pc = strtok (NULL, ":]");
    strncpy (zip, pc, 9);

    *street_ind = *number_ind = *city_ind
    = *state_ind = *zip_ind = *us_addr_ind = 0;
}
```

Step 4. Create a SQL-bodied UDF that chooses the correct external UDF for processing that instance. The following UDF uses the TYPE predicate to invoke the correct to client transform. The results are placed in a temporary table:

```
CREATE FUNCTION stream_address (ENCODING VARCHAR(150))
   RETURNS Address_t
   LANGUAGE SQL
   RETURN
   (CASE(SUBSTR(ENCODING,2,POSSTR(ENCODING,']')-2))
   WHEN 'address_t'
      THEN client_to_address(ENCODING)
   WHEN 'us_addr_t'
      THEN client_to_us_addr(ENCODING)
   ELSE NULL
   END);
```

Step 5. Add the `stream_address` UDF as a TO SQL client transform for `Address_t`:

```
CREATE TRANSFORM FOR Address_t
   client_group (TO SQL
   WITH FUNCTION stream_address);
```

Step 6. Bind the application with the TRANSFORM GROUP option set to `client_group`.

```
PREP myProgram2 TRANSFORM GROUP client_group
```

When the application containing the INSERT statement with a structured type is bound, DB2 looks for a TO SQL client transform. DB2 looks for the transform in the transform group `client_group` because that is the TRANSFORM GROUP specified at bind time in 6. DB2 finds the transform function it needs: `stream_address`, which is associated with the root type `Address_t` in 5.

`stream_address` is a SQL-bodied function, defined in 4, so it has no stated dependency on any additional transform function. For input parameters, `stream_address` accepts VARCHAR(150), which corresponds to the application host variable `:hvaddr`. `stream_address` returns a value that is both of the correct root type, `Address_t`, and of the correct dynamic type.

`stream_address` parses the VARCHAR(150) input parameter for a substring that names the dynamic type: in this case, either 'Address_t' or 'US_addr_t'. `stream_address` then invokes the corresponding external UDF to parse the VARCHAR(150) and returns an object of the specified type. There are two `client_to_address()` UDFs, one to return each possible type. These UDFs are defined in 3 on page 345. Each UDF takes the input VARCHAR(150), and internally constructs the attributes of the appropriate structured type, thus returning the structured type.

To return the structured types, each UDF needs a TO SQL transform function to construct the output attribute values into an instance of the structured type. The CREATE FUNCTION statements in 3 on page 345 name the TRANSFORM GROUP that contains the transforms.

The SQL-bodied transform functions from 1 on page 344, and the associations with the transform groups from 2 on page 345, are named in the CREATE FUNCTION statements of 3 on page 345.

Working with Structured Type Host Variables

Declaring Structured Type Host Variables

To retrieve or send structured type host variables in static SQL, you must provide an SQL declaration that indicates the built-in type used to represent the structured type. The format of the declaration is as follows:

```
EXEC SQL BEGIN DECLARE SECTION ;

   SQL TYPE IS structured_type AS base_type host-variable-name ;

EXEC SQL END DECLARE SECTION;
```

For example, assume that the type `Address_t` is to be transformed to a varying-length character type when passed to the client application. Use the following declaration for the `Address_t` type host variable:

```
SQL TYPE IS Address_t AS VARCHAR(150) addrhv;
```

Describing a Structured Type

A DESCRIBE of a statement with a structured type variable causes DB2 to put a description of the result type of the FROM SQL transform function in the SQLTYPE field of the base SQLVAR of the SQLDA. However, if there is no FROM SQL transform function defined, either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register or because the named group does not have a FROM SQL transform function defined, DESCRIBE returns an error.

The actual name of the structured type is returned in SQLVAR2. For more information about the structure of the SQLDA, refer to the *SQL Reference*.

Chapter 13. Using Large Objects (LOBs)

What are 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.

Note: DB2 offers LOB support for JDBC and SQLJ applications. For more information on using LOBs in Java applications, see "JDBC 2.0" on page 649.

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 "Chapter 11. User-defined Distinct Types" on page 281.

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, refer to 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, refer to 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, refer to 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 443.

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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

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 119 for the source code for this error checking utility.

C Sample: LOBLOC.SQC

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

EXEC SQL INCLUDE SQLCA;

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

   EXEC SQL BEGIN DECLARE SECTION; 1
      char number[7];
      sqlint32 deptInfoBeginLoc;
      sqlint32 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;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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;
   EMB_SQL_CHECK("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;
         EMB_SQL_CHECK("VALUES1");

         /* Locate the beginning of "Education" section (end of "Dept.Info" */
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
         EMB_SQL_CHECK("VALUES2");

         /* Obtain ONLY the "Department Information" section by using SUBSTR */
         EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
         EMB_SQL_CHECK("VALUES3");

         /* Append the "Department Information" section to the :buffer var. */
         EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
         EMB_SQL_CHECK("VALUES4");
      } /* endif */
   } while ( 1 );

   printf ("%s\n",buffer);

   EXEC SQL FREE LOCATOR :resume, :deptBuffer; 3
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CLOSE c1;
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("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.
```

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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 for the source code for this error checking utility.

C Sample: LOBEVAL.SQC

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

EXEC SQL INCLUDE SQLCA;

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

   EXEC SQL BEGIN DECLARE SECTION; 1
      char userid[9];
      char passwd[19];
      sqlint32        hv_start_deptinfo;
      sqlint32        hv_start_educ;
      sqlint32        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;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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
   EMB_SQL_CHECK("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
   EMB_SQL_CHECK("VALUES1");

   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
```

```
      INTO :hv_start_educ;
   EMB_SQL_CHECK("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;
   EMB_SQL_CHECK("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;
   EMB_SQL_CHECK("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;
   EMB_SQL_CHECK("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
   EMB_SQL_CHECK("INSERT");

   printf ("LOBEVAL completed\n");

   /* free the locators */  5
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2, : hv_doc_locator3;
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("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 requester 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 525.
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, refer to 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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

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

See "Using GET ERROR MESSAGE in Example Programs" on page 119 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 "utilemb.h"

EXEC SQL INCLUDE SQLCA;

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;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("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;
   EMB_SQL_CHECK("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);
```

Chapter 14. User-Defined Functions (UDFs) and Methods

What are Functions and Methods?

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. For a complete list of the functions supplied with DB2, refer to the "Supported Functions" table in the *SQL Reference*.

Methods, like UDFs, enable you to write your own extensions to SQL by defining the behavior of SQL objects. However, unlike UDFs, you can only associate a method with a structured type stored as a column in a table.

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 "Chapter 11. User-defined Distinct Types" on page 281 and "Chapter 12. Working with Complex Objects: User-Defined Structured Types" on page 291, distinct types and structured types can be very useful in extending the capability and increasing the safety of DB2. You can create methods that define the behavior for structured types stored in columns. You can also create functions that act on distinct types.

Why Use Functions and Methods?

In writing DB2 applications, you have a choice when implementing desired actions or operations:

- As a UDF
- As a method
- As a subroutine in your application.

Although it may seem easier to implement new operations as subroutines in your application, there are good reasons why you should consider using UDFs and methods:

- **Re-use**.

  If the new operation is something of which other users or programs at your site can take advantage, then UDFs and methods can help to reuse it. In addition, the operation can be invoked directly in SQL wherever an expression can be used by any user of the database. For UDFs, the database will take care of many data type promotions of the function arguments automatically, for example DECIMAL to DOUBLE, allowing your operation to be applied to different, but compatible data types.

  It may seem easier to implement your new operation as a subroutine 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 subroutine 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 or method in the database. This also applies to any other tools that use SQL (such as Lotus Approach), that can not be recompiled.

- **Performance**.

  Invoking the UDF or method directly from the database engine instead of from your application can have a considerable performance advantage, particularly when the operation qualifies 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 359 for an example of a similar scenario.)

In addition, you can use the RETURNS TABLE clause of the CREATE FUNCTION statement to define UDFs called *table functions*. Table functions enable you to 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, refer to the *SQL Reference*.

You cannot create a method that returns a table.

- **Behavior of Distinct Types**.

  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 "Chapter 11. User-defined Distinct Types" on page 281. For additional details on UDTs and the important concept of *castability* discussed therein, refer to 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 0 if 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 And Method Concepts

The following is a discussion of the important concepts you need to know prior to coding UDFs and methods:

- 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 identify 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, refer to 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 and methods 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 381.

- 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 and methods 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 and return the data as a table. 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*.

For information about the concept of mapping UDFs and methods and built-in functions to data source functions in a federated system, refer to the *SQL Reference*. For guidelines on creating such mappings, refer to "Invoking Data Source Functions" on page 586.

Implementing Functions and Methods

The process of implementing an external UDF or method requires the following steps:

1. Writing the UDF or method
2. Compiling the UDF or method

3. Linking the UDF or method
4. Debugging the UDF or method
5. Registering the UDF or method with DB2

After these steps are successfully completed, your UDF or method is ready for use in DML or DDL statements such as CREATE VIEW. The steps of writing and defining UDFs and methods are discussed in the following sections, followed by a discussion on using UDFs and methods. For information on compiling and linking UDFs and methods, refer to the *Application Building Guide*. For information on debugging your UDF or method, see "Debugging your UDF" on page 480.

Writing Functions and Methods

You can find the details of how you write UDFs and methods in "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393. This includes the details on the interface between DB2 and a UDF or method, coding considerations, coding examples, and debugging information. For information on the related tasks of compiling and linking your UDFs and methods, refer to the *Application Building Guide*.

Registering Functions and Methods

You should register the UDF or method to DB2 after you have written and completely tested the actual code. Note that it is possible to define the UDF or method prior to actually writing it. However, to avoid any problems with running your UDF or method, you are encouraged to write and test it extensively before registering it. For information on testing your UDFs and methods, see "Debugging your UDF" on page 480.

Use the CREATE FUNCTION statement to define (or register) your UDF to DB2. To register a method with DB2, use the CREATE TYPE or ALTER TYPE statement to define a method for a structured type, then use the CREATE METHOD statement to associate the method body with the method specification. You can find detailed explanations for these statements and their syntax in the *SQL Reference*.

Examples of Registering UDFs and Methods

The examples which follow illustrate a variety of typical situations where UDFs and methods can be registered. The examples include:

- Example: Exponentiation
- Example: String Search
- Example: String Search over UDT
- Example: External Function with UDT Parameter
- Example: AVG over a UDT

- Example: Counting

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 or CREATE METHOD 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 UNIX platforms. 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. Refer to 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 382). 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 and methods 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 381, 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 381. 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 381, 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 381. 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 euros, 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 an exchange rate web page, and the answer depends on the contents of the web page. 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 283 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 default size of the scratchpad (100 bytes) 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 461.

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 474 and in "Example: Counter OLE Automation UDF in C++" on page 476. For details of OLE support with DB2, see "Writing OLE Automation UDFs" on page 425.

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
  NO 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. FINAL CALL does not need to 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.

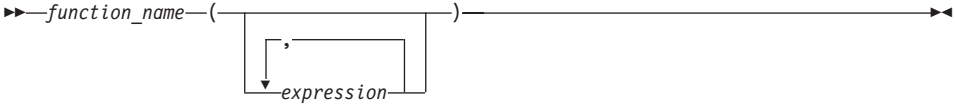
Using Functions and Methods

Scalar and column UDFs and methods 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.

See "UDF And Method Concepts" on page 377 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:



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, refer to 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, DOUBLE, REAL, or BIGINT, 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, refer to 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 387, 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 387, 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 453 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 and methods 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 380, 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 443 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 and methods 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 382, 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.

Chapter 15. Writing User-Defined Functions (UDFs) and Methods

Description

This section describes how to write UDFs and methods. The coding conventions for UDFs and methods are the same, with the following differences:

- Since DB2 associates each method with a specific structured type, the first argument passed from DB2 to your method is always the instance of the structured type on which you invoked the method.
- Methods, unlike UDFs, cannot return tables. You cannot invoke a method as the argument for a FROM clause.

As the guidelines for writing UDFs and methods are the same, with the exception of the previously described difference, the remainder of the discussion on writing UDFs and methods refers to both UDFs and methods simply as UDFs.

For small UDFs such as UDFs that contain only simple logic, consider using a *SQL-bodied UDF*. To create a SQL-bodied UDF, issue a CREATE FUNCTION or CREATE METHOD statement that includes a method body written using SQL, rather than pointing to an external UDF. SQL-bodied UDFs enable you to declare and define the UDF in a single step, without using an external language or compiler. SQL-bodied UDFs also offer the possibility of increased performance, because the method body is written using SQL accessible to the DB2 optimizer.

The following example demonstrates a simple CREATE FUNCTION statement that creates a SQL-bodied UDF:

```
CREATE FUNCTION tan(double x)
RETURNS double
NO EXTERNAL ACTION
DETERMINISTIC
LANGUAGE SQL
CONTAINS SQL
RETURN sin(x) / cos(x);
```

For further information on SQL-bodied functions, refer to the *SQL Reference*.

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:

- Important material on defining and using UDFs is presented in "Chapter 14. User-Defined Functions (UDFs) and Methods" on page 373, and is not repeated here. This discussion concentrates on how you implement a UDF.
- To implement an *external UDF* written in C, C++ or Java, you must perform the following steps:
  - Write the UDF
  - Compile the UDF
  - Link the UDF
  - Register the UDF with a CREATE FUNCTION statement
  - Test and debug the UDF

  You can find information on compiling and linking UDFs in the *Application Building Guide*.

- You can invoke your UDF using OLE (Object Linking and Embedding) as described in "Writing OLE Automation UDFs" on page 425.
- You can define an *OLE DB table function*, which is a function that returns a table from an OLE DB data source, with just a CREATE FUNCTION statement. For more information on OLE DB table functions, see "OLE DB Table Functions" on page 431.

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.

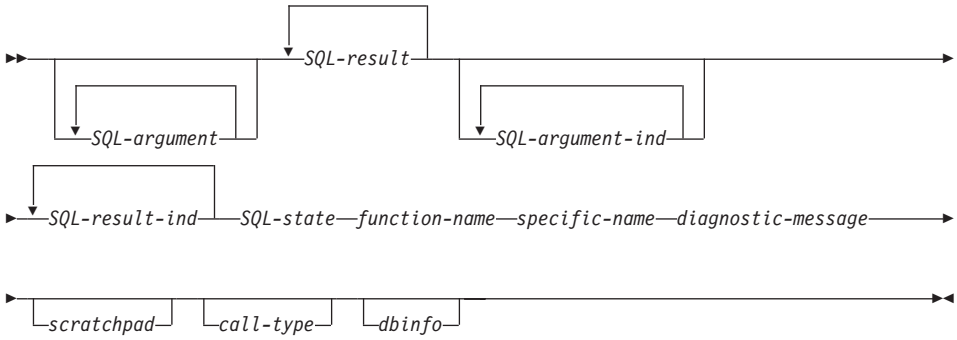
Interface between DB2 and a UDF

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 420.

The Arguments Passed from DB2 to a UDF

In addition to the SQL arguments which are specified 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 "Passing Arguments to a UDF" on page 396. Java UDFs take only the *SQL-argument* and *SQL-result* arguments, but can call extra methods to access the other information. 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 420.

Syntax for 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 corresponding parameter definition in the CREATE FUNCTION statement. How these data types map to C language constructs is described in "How the SQL Data Types are Passed to a UDF" on page 410.

DB2 aligns the data for the *SQL-argument* according to the data type and the server platform.

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 by position to 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 410.

Note that for table functions, DB2 defines a performance optimization where every defined column does not have to be returned to DB2. If you write your UDF to take advantage of this feature, it 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 return only those two columns for each row and not spend time on the other 98 columns. 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 return 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 that DB2 does not detect can cause unpredictable results or an abnormal termination.

DB2 aligns the data for *SQL-result* according to the data type and the server platform.

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 410. DB2 aligns the data for *SQL-argument-ind* according to the data type and the server platform.

SQL-result-ind

This argument is set by the UDF before returning to DB2. There is one of these for each *SQL-result* argument.

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. For more information, see "Interpreting Negative SQL-result-ind Values".

Interpreting Negative SQL-result-ind Values:

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
- The `SQL-result-ind` is negative.

This is also true if you define the function with the NOT NULL CALL option.

Even if the function is defined with NOT NULL CALL, the UDF body must set the indicator of the result. For example, a divide function could set the result to null when the denominator is zero.

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 410.

If the UDF takes advantage of table function optimization using the RESULT column list, then only the indicators corresponding to the required columns need be set.

DB2 aligns the data for *SQL-result-ind* according to the data type and the server platform.

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 410, 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 the FETCH call to 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).

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 410 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 and schema 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 410 for more information. Two examples are:

```
willie_find_feb99       SQL9904281052440430
```

This first value is provided by the user in his CREATE FUNCTION statement. The second value is generated by DB2 from the current timestamp if 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 410 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 part of 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, because DB2 may truncate the tokens to conform to the restrictive limit on total token length imposed by the SQLCA. Avoid using X'FF' in the text since this character is used to delimit tokens in the SQLCA.

The UDF code should not return more text than will fit in 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. Changing the length of the scratchpad will result in SQLCODE -450 (SQLSTATE 39501)
- The actual scratchpad initialized to all binary 0's as follows:

  For scalar functions it is initialized before the first call, and not generally looked at or modified by DB2 thereafter.

  For table functions, the scratchpad is initialized as above prior to the FIRST call to the UDF if FINAL CALL is specified on the CREATE FUNCTION. After this call, the scratchpad content is totally under control of the table function.

  If NO FINAL CALL was specified or defaulted for a table function, then the scratchpad is initialized as above for each OPEN call, and the scratchpad content is completely under control of the table function between OPEN calls. (This can be very important for a table function used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH and CLOSE calls, the table function will also receive FIRST and FINAL calls, for the purpose of scratchpad maintenance and resource release.)

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 410 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.

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.

DB2 initializes the scratchpad so that the data field is aligned for the storage of any data type. This may result in the entire scratchpad structure, including the length field, not being properly aligned. For more information on declaring and accessing scratchpads, see "Writing Scratchpads on 32-bit and 64-bit Platforms" on page 418.

call-type

This argument, if present, is set by DB2 before calling the UDF. For scalar functions this argument is only present if FINAL CALL is specified in the CREATE FUNCTION statement, but for table functions it is ALWAYS present. 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 410 for more information.

Note that even though all the current possible values are listed below, your UDF should contain a switch or case statement which explicitly tests for all the expected values, rather than containing "if A do AA, else if B do BB, else it must be C so do CC" type logic. This is because it is possible that additional call types may be added in the future, and if you don't explicitly test for condition C you will have trouble when new possibilities are added.

Notes:

1. For all the call-types, it may be appropriate for the UDF to set a *SQL-state* and *diagnostic-message* return value. This information will not be repeated in the following descriptions of each call-type. For all calls DB2 will take the indicated action as described previously for these arguments.
2. The include file `sqludf.h` is intended for use with UDFs and is described in "The UDF Include File: sqludf.h" on page 419. The file contains symbolic defines for the following call-type values, which are spelled out as constants.

For scalar functions *call-type* contains:

-1 This is the *FIRST 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. In addition, a FIRST call to a scalar UDF is like a NORMAL call, in that it is expected to develop and return an answer.

Note that if SCRATCHPAD is specified but FINAL CALL is not, then the UDF will not have this call-type argument to identify the very first call. Instead it will have to rely on the all-zero state of the scratchpad.

0 This is a *NORMAL call*. All the SQL input values are passed, and the UDF is expected to develop and return the result. The UDF may also return *SQL-state* and *diagnostic-message* information.

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. The UDF is expected to release resources at this point.

Releasing resources.

A scalar UDF is expected to release resources it has required, for example, memory. If FINAL CALL is specified for the UDF, then that FINAL call is a natural place to release resources, provided that SCRATCHPAD is also specified and is used to track the resource. If FINAL CALL is not specified, then any resource acquired should be released on the same call.

For table functions *call-type* contains:

-2 This is the *FIRST call*, which only occurs if the FINAL CALL keyword was specified for the UDF. The scratchpad is set to binary zeros before this call. Argument values are passed to the table function, and it may choose to acquire memory or perform other one-time only resource initialization. Note that this is not an OPEN call, that call follows this one. On a FIRST call the table function should not return any data to DB2 as DB2 ignores the data.

-1 This is the *OPEN call*. The scratchpad will be initialized if NO FINAL CALL is specified, but not necessarily otherwise. All SQL argument values are passed to the table function on OPEN. The table function should not return any data to DB2 on the OPEN call.

0 This is a FETCH call, and DB2 expects the table function to return either a row comprising the set of return values, or an end-of-table condition indicated by SQLSTATE value '02000'. If a scratchpad is passed to the UDF, then on entry it is untouched from the previous call.

1 This is a CLOSE call to the table function. It balances the OPEN call, and can be used to perform any external CLOSE processing (for example, closing a source file), and resource release (particularly for the NO FINAL CALL case).

In cases involving a join or a subquery, the OPEN/FETCH.../CLOSE call sequences can repeat within the execution of a statement, but there is only one FIRST call and only one FINAL call. The FIRST and FINAL call only occur if FINAL CALL is specified for the table function.

2 This is a FINAL call, which only occurs if FINAL CALL was specified for the table function. It balances the FIRST call, and occurs only once per execution of the statement. It is intended for the purpose of releasing resources.

Releasing resources.

Write UDFs to release any resources that they acquire. For table functions, there are two natural places for this release: the CLOSE call and the FINAL call. The CLOSE call balances each OPEN call and can occur multiple times in the execution of a statement. The FINAL call only occurs if FINAL CALL is specified for the UDF, and occurs only once per statement.

If you can apply a resource across all OPEN/FETCH/CLOSE sequences of the UDF, write the UDF to acquire the resource on the FIRST call and free it on the FINAL call. The scratchpad is a natural place to track this resource. For table functions, if FINAL CALL is specified, the scratchpad is initialized only before the FIRST call. If FINAL CALL is not specified, then it is reinitialized before each OPEN call.

If a resource is specific to each OPEN/FETCH/CLOSE sequence, write the UDF to free the resource on the CLOSE call. (Note that when a table function is in a subquery or join, it is very possible that there will be multiple occurrences of the OPEN/FETCH/CLOSE sequence, depending on how the DB2 Optimizer chooses to organize the execution of the statement.)

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 the `sqludf_dbinfo` structure defined in the header file `sqludf.h`, which is discussed in "The UDF Include File: sqludf.h" on page 419. The variables in this structure that contain names and identifiers may be longer than the longest value possible in this release of DB2, but are defined this way for compatibility with future releases. You can use the length variable that complements each name and identifier variable to read or extract the portion of the variable that is actually used. The *dbinfo* structure 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 a union of two 48-byte long structures; one is used by DB2 Universal Database, the other is reserved for future use. The structure used by DB2 Universal Database 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.

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 Universal Database 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 operating platform for the application server, as follows:
    SQLUDF_PLATFORM_AIX AIX
    SQLUDF_PLATFORM_HP HP-UX
    SQLUDF_PLATFORM_MVS OS/390
    SQLUDF_PLATFORM_NT Windows NT
    SQLUDF_PLATFORM_OS2 OS/2
    SQLUDF_PLATFORM_SUN Solaris Operating Environment
    SQLUDF_PLATFORM_WINDOWS
    Windows 95 and Windows 98

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 2 characters long.

16. Procedure ID (procid)

    The value of the procid field in the DBINFO structure that is being passed to a procedure or function is non-zero if the caller of the routine is a cataloged stored procedure. In this situation, the value of procid is the ID of the calling procedure as recorded in the PROCEDURE_ID column of the SYSCAT.PROCEDURES table. In all other situations, the value returned for the procid field is 0.

17. Reserved field (resd2)

    This field is for future use. It is defined as 32 characters long.

18. Table function column list (tfcolumn)

    If this is a table function, this field is a pointer to an array of short integers which is dynamically allocated by DB2. If this is a scalar function, 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, `numtfcol`, 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 ordinal 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 variable `numtfcol` might be zero, for a statement similar to `SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ`, where no actual column values are needed by the query.

    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 choose to return every defined column.

19. Unique application identifier (appl_id)

    This field is a pointer to a C null-terminated string which uniquely identifies the application's connection to DB2. It is regenerated at each database connect.

    The string has a maximum length of 32 characters, and its exact format depends on the type of connection established between the client and DB2. Generally it takes the form

    ```
    <x>.<y>.<ts>
    ```

    where the <x> and <y> vary by connection type, but the <ts> is a 12 character time stamp of the form YYMMDDHHMMSS, which is potentially adjusted by DB2 to ensure uniqueness.

    ```
    Example:  *LOCAL.db2inst.980707130144
    ```

20. Reserved field (resd3)

    This field is for future use. It is defined as 20 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, the arguments 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 and FIRST calls. 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, and optionally FIRST and FINAL.

- *dbinfo*.

  This is a structure passed by DB2 to the UDF containing additional information.

Observe that the normal value outputs of the UDF, as well as the *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 supported operating systems. This macro is required when you write stored procedures or UDFs.

For programming examples of UDFs, see "Examples of UDF Code" on page 453.

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. For type definitions in Java UDFs, see "Supported SQL Data Types in Java" on page 639. 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 and compilers. For example, for BIGINT you can use the SQLUDF_BIGINT data type to hide differences in the name of the 64 bit integer type between different compilers. This include file is discussed in "The UDF Include File: sqludf.h" on page 419.

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. For a list of SQL type representations for Java, see "Supported SQL Data Types in Java" on page 639. 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)...
  ```

  Example:

  ```
  short    *arg1;          /* example for SMALLINT */
  short    *arg1_null_ind; /* example for any null indicator */
  ```

  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.

- INTEGER or INT

  Valid. Represent in C as `sqlint32`. The DB2 include file `sqlsystm.h` defines this type as the appropriate 32-bit integer for your platform.

  Example:

```
sqlint32 *arg2;          /* example for INTEGER */
```

- BIGINT

  Valid. Represent in C as `sqlint64`.

  Example:

  ```
  sqlint64 *arg3;          /* example for INTEGER */
  ```

  DB2 defines the `sqlint64` C language type to overcome differences between definitions of the 64 bit signed integer in compilers and operating systems. You must `#include sqludf.h` to pick up the definition.

- 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').

  Example:

```
char     arg1[14];      /* example for CHAR(13)   */
char    *arg1;          /* also perfectly acceptable */
```

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.

- 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. See "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622 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 in C and C++" on page 623. 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. See "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622 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 in C and C++" on page 623. 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. See "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622 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 in C and C++" on page 623. 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
  {
      sqluint32     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
  {
      sqluint32 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. See "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622 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 in C and C++" on page 623. 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 419. The use of these locators is discussed in "Using LOB Locators as UDF Parameters or Results" on page 443.

Writing Scratchpads on 32-bit and 64-bit Platforms

To make your UDF code portable between 32-bit and 64-bit platforms, you must change the way in which you create and use scratchpads that contain 64-bit values. Do not declare an explicit length variable for a scratchpad structure that contains one or more 64-bit values, such as 64-bit pointers or `sqlint64` BIGINT variables. For example, the following example might result

in a data alignment exception on a 64-bit platform because the structure declaration includes an explicit length variable:

```
struct scratch1
{
  sqlint32 length;
  char chars[4];
  sqlint64 bigint_var;
};
```

To declare the scratchpad structure from the previous example so that it is portable between 32-bit and 64-bit platforms, remove the declaration of the explicit length variable for the structure. The following example declares the scratchpad structure without declaring an explicit length variable:

```
struct scratch1
{
  sqlint64 bigint_var;
  char chars[4];
};
```

To access a scratchpad structure that does not declare an explicit length variable in your UDF, you can refer to the scratchpad using the following format:

```
struct scratchpad_data * data =
  (struct scratchpad_data*)scratch_pointer->data;
```

where *scratch_pointer* represents the *sqludf_scratchpad* pointer of the UDF and *data* represents the contents of the scratchpad.

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 453, 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 395. 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`.

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. Refer to the *SQL Reference* for information on registering a Java UDF using this statement. You can then refer to it in the SQL of your application. The UDF can be FENCED or NOT FENCED, and you can also use options to modify how the UDF is run. See "Changing How a Java UDF Runs" on page 422.

Some sample Java UDF method bodies are provided in the UDFsrv.java sample. You can find the associated CREATE FUNCTION statements and examples of calling those UDFs in the UDFcli.java and UDFclie.sqlj samples. See the sqllib/samples/java directory for the samples and README instructions for compiling and running the samples.

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 extends UDF {
   public void test3(String arg1, Blob arg2, String arg3,
                     int result) { ... }
}
```

Java UDFs that implement table functions require 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` class.

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 664.

Changing How a Java UDF Runs

Typically, DB2 calls a UDF many times, once for each row of an input or result set in a query. If SCRATCHPAD is specified in the CREATE FUNCTION statement of the UDF, DB2 recognizes that some "continuity" is needed between successive invocations of the UDF, and therefore the implementing Java class is not instantiated for each call, but generally speaking once per UDF reference per statement. Generally it is instantiated before the first call and used thereafter, but may for table functions be instantiated more often. For more information, see the NO FINAL CALL execution model in the subsection which follows this one.

If, however, NO SCRATCHPAD is specified for a UDF, either a scalar or table function, then a clean instance is instantiated for each call to the UDF.

A scratchpad may be useful for saving information across calls to a UDF. While Java and OLE UDFs can either use instance variables or set the scratchpad to achieve continuity between calls, C and C++ UDFs must use the scratchpad. Java UDFs access the scratchpad with the getScratchPad() and setScratchPad() methods available in COM.ibm.db2.app.UDF.

For Java table functions that use a scratchpad, control when you get a new scratchpad instance by using the FINAL CALL or NO FINAL CALL option on the CREATE FUNCTION statement, as indicated by the execution models in "Table Function Execution Model for Java" on page 423.

The ability to achieve continuity between calls to a UDF by means of a scratchpad is controlled by the SCRATCHPAD and NO SCRATCHPAD option of CREATE FUNCTION, regardless of whether the DB2 scratchpad or instance variables are used.

For scalar functions, you use the same instance for the entire statement.

Please note that every reference to a Java UDF in a query is treated independently, even if the same UDF is referenced multiple times. This is the same as what happens for OLE, C and C++ UDFs as well. At the end of a query, if you specify the FINAL CALL option for a scalar function then the object's close() method is called. For table functions the close() method will always be invoked as indicated in the subsection which follows this one. If you do not define a close() method for your UDF class, then 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.

As with other UDFs, Java UDFs can be FENCED or NOT FENCED. NOT FENCED UDFs run inside the address space of the database engine; FENCED UDFs 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 debug UDFs written in Java, you should run them as FENCED UDFs.

See "COM.ibm.db2.app.UDF" on page 773 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`.

Table Function Execution Model for Java

For table functions written in Java, it is important to understand what happens at each point in DB2's processing of a given statement which is significant to the table function. The table which follows details this information. The bottom part of each box hints what the code might be written to do for a typical table function which pulls some information in from the Web. Covered are both the NO FINAL CALL and the FINAL CALL cases, assuming SCRATCHPAD in both cases.

Point in scan time	**NO FINAL CALL LANGUAGE JAVA SCRATCHPAD**	**FINAL CALL LANGUAGE JAVA SCRATCHPAD**
Before the first OPEN for the table function	No calls.	• Class constructor is called (means new scratchpad). UDF method is called with FIRST call. • Constructor initializes class and scratchpad variables. Method connects to Web server.

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
At each OPEN of the table function	• Class constructor is called (means new scratchpad). UDF method is called with OPEN call. • Constructor initializes class and scratchpad variables. Method connect to Web server, and opens the scan for Web data.	• UDF method is opened with OPEN call. • Method opens the scan for whatever Web data it wants. (Might be able to avoid reopen after a CLOSE reposition, depending on what is saved in the scratchpad.)
At each FETCH for a new row of table function data	• UDF method is called with FETCH call. • Method fetches and returns next row of data, or EOT.	• UDF method is called with FETCH call. • Method fetches and returns new row of data, or EOT.
At each CLOSE of the table function	• UDF method is called with CLOSE call. `close()` method if it exists for class. • Method closes its Web scan and disconnects from the Web server. `close()` does not need to do anything.	• UDF method is called with CLOSE call. • Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist.
After the last CLOSE of the table function	No calls.	• UDF method is called with FINAL call. `close()` method is called if it exists for class. • Method disconnects from the Web server. `close()` method does not need to do anything.

Notes:

1. By "UDF method" we mean the Java class method which implements the UDF. This is the method identified in the EXTERNAL NAME clause of the CREATE FUNCTION statement.
2. For table functions with NO SCRATCHPAD specified, the calls to the UDF method are as indicated in this table, but because the user is not asking for any continuity via a scratchpad, DB2 will cause a new object to be instantiated before each call, by calling the class constructor. It is not clear that table functions with NO SCRATCHPAD (and thus no continuity) can do very useful things, but they are supported.
3. These models are TOTALLY COMPATIBLE with what happens with the other UDF languages: C/C++ and OLE.

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. Other applications, such as Lotus Notes or Microsoft Exchange®, can then integrate these objects by taking advantage of these properties and methods 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.

For a list of sample applications included with the DB2 Application Development Client that demonstrate OLE automation UDFs, see Table 50 on page 761.

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, but you must use the following options:

- LANGUAGE OLE
- FENCED, since OLE automation UDFs must run in FENCED mode

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 17 on page 428.

Table 16. 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 BIGINT, 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. This section shows you how to implement OLE automation UDFs using BASIC or C++ as two sample languages.

Table 17 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 16 on page 427 holds true for all languages).

Table 17. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types

SQL Type	OLE Automation Type	UDF Language	
		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 bcounter OLE automation UDF in "Creating and Registering OLE Automation UDFs" on page 425 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)
```

You can find an example of an OLE table automation in "Example: Mail OLE Automation Table Function in BASIC" on page 478.

OLE Automation UDFs in C++

Table 17 on page 428 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 increment 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.

OLE DB Table Functions

Microsoft OLE DB is a set of OLE/COM interfaces that provide applications with uniform access to data stored in diverse information sources. The OLE DB component DBMS architecture defines OLE DB consumers and OLE DB providers. An OLE DB consumer is any system or application that consumes OLE DB interfaces; an OLE DB provider is a component that exposes OLE DB interfaces. There are two classes of OLE DB providers: *OLE DB data providers*, which own data and expose their data in tabular format as a rowset; and *OLE DB service providers*, which do not own their own data, but encapsulate some services by producing and consuming data through OLE DB interfaces.

DB2 Universal Database simplifies the creation of OLE DB applications by enabling you to define table functions that access an OLE DB data source. DB2 is an OLE DB consumer that can access any OLE DB data or service provider. You can perform operations including GROUP BY, JOIN, and UNION on data sources that expose their data through OLE DB interfaces. For example, you can define an OLE DB table function to return a table from a Microsoft Access database or a Microsoft Exchange address book, then create a report that seamlessly combines data from this OLE DB table function with data in your DB2 database.

Using OLE DB table functions reduces your application development effort by providing built-in access to any OLE DB provider. For C, Java, and OLE automation table functions, the developer needs to implement the table function, whereas in the case of OLE DB table functions, a generic built-in OLE DB consumer interfaces with any OLE DB provider to retrieve data. You only need to register a table function of language type OLEDB, and refer to

the OLE DB provider and the relevant rowset as a data source. You do not have to do any UDF programming to take advantage of OLE DB table functions.

To use OLE DB table functions with DB2 Universal Database, you must install OLE DB 2.0 or later, available from Microsoft at `http://www.microsoft.com`. If you attempt to invoke an OLE DB table function without first installing OLE DB, DB2 issues SQLCODE 465, SQLSTATE 58032, reason code 35. For the system requirements and OLE DB providers available for your data sources, refer to your data source documentation. For a list of samples that define and use OLE DB table functions, see "Appendix B. Sample Programs" on page 743. For the OLE DB specification, see the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Creating an OLE DB Table Function

To define an OLE DB table function with a single CREATE FUNCTION statement, you must:

- define the table that the OLE DB provider returns
- specify LANGUAGE OLEDB
- identify the OLE DB rowset and provide an OLE DB provider connection string in the EXTERNAL NAME clause

OLE DB data sources expose their data in tabular form, called *rowset*. A rowset is a set of rows, each having a set of columns. The RETURNS TABLE clause includes only the columns relevant to the user. The binding of table function columns to columns of a rowset at an OLE DB data source is based on column names. If the OLE DB provider is case sensitive, place the column names in quotation marks; for example, `"UPPERcase"`. For information on rowset names which can be fully qualified names, see "Fully Qualified Rowset Names" on page 434. For the mapping of OLE DB data types to DB2 data types, see "Supported OLE DB Data Types" on page 436. For the complete syntax of the CREATE FUNCTION statement and the rules for the EXTERNAL NAME clause, refer to the *SQL Reference*.

The EXTERNAL NAME clause can take either of the following forms:

```
    'server!rowset'
  or
    '!rowset!connectstring'
```

where:

server identifies a server registered with CREATE SERVER statement

rowset
identifies a rowset, or table, exposed by the OLE DB provider; this value should be empty if the table has an input parameter to pass through command text to the OLE DB provider.

connectstring
contains initialization properties needed to connect to an OLE DB provider. For the complete syntax and semantics of the connection string, see the "Data Link API of the OLE DB Core Components" in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

You can use a *connection string* in the EXTERNAL NAME clause of a CREATE FUNCTION statement, or specify the *CONNECTSTRING* option in a CREATE SERVER statement.

For example, you can define an OLE DB table function and return a table from a Microsoft Access database with the following CREATE FUNCTION and SELECT statements:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
                 Data Source=c:\msdasdk\bin\oledb\nwind.mdb';

SELECT orderid, DATE(orderdate) AS orderdate,
                DATE(shippeddate) AS shippeddate
FROM TABLE(orders()) AS t
WHERE orderid = 10248;
```

Instead of putting the connection string in the EXTERNAL NAME clause, you can create and use a server name. For example, assuming you have defined the server `Nwind` as described in "Defining a Server Name for an OLE DB Provider" on page 435, you could use the following CREATE FUNCTION statement:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

OLE DB table functions also allow you to specify one input parameter of any character string data type. Use the input parameter to pass command text directly to the OLE DB provider. If you define an input parameter, do not provide a rowset name in the EXTERNAL NAME clause. DB2 passes the command text to the OLE DB provider for execution and the OLE DB provider returns a rowset to DB2. Column names and data types of the resulting rowset need to be compatible with the RETURNS TABLE definition in the CREATE FUNCTION statement. Since binding to the column names of the rowset is based on matching column names, you must ensure that you name the columns properly.

The following example registers an OLE DB table function, which retrieves store information from a Microsoft SQL Server 7.0™ database. The connection

string is provided in the EXTERNAL NAME clause. Since the table function has an input parameter to pass through command text to the OLE DB provider, the rowset name is not specified in the EXTERNAL NAME clause. The query example passes in a SQL command text which retrieves information about the top three stores from a SQL Server database.

```
CREATE FUNCTION favorites (varchar(600))
  RETURNS TABLE (store_id char (4), name varchar (41), sales integer)
  SPECIFIC favorites
  LANGUAGE OLEDB
  EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
  User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
  Locale Identifier=1033;Use Procedure for Prepare=1;
  Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
  OLE DB Services=CLIENTCURSOR;';

SELECT *
FROM TABLE (favorites (' select top 3 sales.stor_id as store_id,  ' ||
                       '        stores.stor_name as name,         ' ||
                       '        sum(sales. qty) as sales          ' ||
                       ' from sales, stores                       ' ||
                       ' where sales.stor_id = stores.stor_id     ' ||
                       ' group by sales.stor_id, stores.stor_name ' ||
                       ' order by sum(sales.qty) desc')) as f;
```

Fully Qualified Rowset Names

Some rowsets need to be identified in the EXTERNAL NAME clause through a *fully qualified name*. A fully qualified name incorporates either or both of the following:

- the associated catalog name, which requires the following information:
  - whether the provider supports catalog names
  - where to put the catalog name in the fully qualified name
  - which catalog name separator to use
- the associated schema name, which requires the following information:
  - whether the provider supports schema names
  - which schema name separator to use

For information on the support offered by your OLE DB provider for catalog and schema names, refer to the documentation of the literal information of your OLE DB provider.

If DBLITERAL_CATALOG_NAME is not NULL in the literal information of your provider, use a catalog name and the value of DBLITERAL_CATALOG_SEPARATOR as a separator. To determine whether the catalog name goes at the beginning or the end of the fully qualified name, refer to the value of DBPROP_CATALOGLOCATION in the property set DBPROPSET_DATASOURCEINFO of your OLE DB provider.

If DBLITERAL_SCHEMA_NAME is not NULL in the literal information of your provider, use a schema name and the value of DBLITERAL_SCHEMA_SEPARATOR as a separator.

If the names contain special characters or match keywords, enclose the names in the quote characters specified for your OLE DB provider. The quote characters are defined in the literal information of your OLE DB provider as DBLITERAL_QUOTE_PREFIX and DBLITERAL_QUOTE_SUFFIX. For example, in the following EXTERNAL NAME the specified rowset includes catalog name *pubs* and schema name *dbo* for a rowset called *authors*, with the quote character " used to enclose the names.

```
EXTERNAL NAME '!"pubs"."dbo"."authors"!Provider=SQLOLEDB.1;...';
```

For more information on constructing fully qualified names, refer to *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998, and the documentation for your OLE DB provider.

Defining a Server Name for an OLE DB Provider

Before you can define a server name for an OLE DB provider, you must register the OLE DB wrapper once per database using the CREATE WRAPPER OLEDB statement. For more information on CREATE WRAPPER OLEDB, refer to the *Installation and Configuration Supplement*.

To provide a server name for an OLE DB data source that you can use for many CREATE FUNCTION statements, use the CREATE SERVER statement as follows:

- provide a name that identifies the OLE DB provider within DB2
- specify WRAPPER OLEDB
- provide connection information in the CONNECTSTRING option

For example, you can define the server name Nwind for the Microsoft Access OLE DB provider with the following CREATE SERVER statement:

```
CREATE SERVER Nwind
  WRAPPER OLEDB
  OPTIONS (CONNECTSTRING 'Provider=Microsoft.Jet.OLEDB.3.51;
           Data Source=c:\msdasdk\bin\oledb\nwind.mdb');
```

You can then use the server name Nwind to identify the OLE DB provider in a CREATE FUNCTION statement, for example:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

For the complete syntax of the CREATE SERVER statement, refer to the *SQL Reference*. For information on user mappings for OLE DB providers, see "Defining a User Mapping".

Defining a User Mapping

You can provide *user mappings* for your DB2 users to provide access to OLE DB data sources with an alternate username and password. To map usernames for specific users, you can define user mappings with the CREATE USER MAPPING statement. To provide a user mapping shared by all users, add the username and password to the connection string of your CREATE FUNCTION or CREATE SERVER statement. For example, to create a specific user mapping for the DB2 user `JOHN` on the OLE DB server `Nwind`, use the following CREATE USER MAPPING statement:

```
CREATE USER MAPPING FOR john
  SERVER Nwind
  OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');
```

To provide the equivalent access to all of the DB2 users that call the OLE DB table function `orders`, use the following CONNECTSTRING either in a CREATE FUNCTION or CREATE SERVER statement:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;User ID=dave;
                 Password=mypwd;Data Source=c:\msdasdk\bin\oledb\nwind.mdb';
```

DB2 applies the following user mapping rules:

- If a user mapping is defined, DB2 uses the mapped authorization information to connect to the OLE DB provider, potentially overwriting an existing user ID and password in the CONNECTSTRING.
- If no user mapping is defined, DB2 uses the authorization information from the CONNECTSTRING, if the information is provided.
- If no user mapping is defined and no authorization information is provided in the CONNECTSTRING, DB2 uses the current DB2 authorization information if the provider supports authorization.

For the complete syntax of the CREATE USER MAPPING statement, refer to the *SQL Reference*.

Supported OLE DB Data Types

The following table shows how DB2 data types map to the OLE DB data types described in *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. Use the mapping table to define the appropriate RETURNS TABLE columns in your OLE DB table functions. For example, if you define an OLE DB table function with a column of data type INTEGER, DB2 requests the data from the OLE DB provider as DBTYPE_I4.

For mappings of OLE DB provider source data types to OLE DB data types, refer to the OLE DB provider documentation. For examples of how the ANSI SQL, Microsoft Access, and Microsoft SQL Server providers might map their respective data types to OLE DB data types, refer to the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Table 18. Mapping DB2 Data Types to OLE DB

DB2 Data Type	OLE DB Data Type
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

Note: OLE DB data type conversion rules are defined in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. For example:

- To retrieve the OLE DB data type DBTYPE_CY, the data may get converted to OLE DB data type DBTYPE_NUMERIC(19,4) which maps to DB2 data type DEC(19,4).
- To retrieve the OLE DB data type DBTYPE_I1, the data may get converted to OLE DB data type DBTYPE_I2 which maps to DB2 data type SMALLINT.
- To retrieve the OLE DB data type DBTYPE_GUID, the data may get converted to OLE DB data type DBTYPE_BYTES which maps to DB2 data type CHAR(12) FOR BIT DATA.

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 448.

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 461.

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. If the table function is specified with NO FINAL CALL (the default), DB2 refreshes the scratchpad before each OPEN call. If you specify the table function option FINAL CALL, DB2 does not examine or change the content of the scratchpad thereafter. The scratchpad is passed to the function on each invocation. The function can be re-entrant, and DB2 preserves its state information in the scratchpad.

So for the *counter* example, the last value returned could be kept in the scratchpad. And the *match* example 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.

If you do specify FINAL CALL, please note that your UDF receives a call of type FIRST. This could be used to acquire and initialize some persistent resource.

Table Function Considerations

An external table function is a 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 five types of call made to a table function: OPEN, FETCH, CLOSE, FIRST, and FINAL. The existence of FIRST and FINAL calls depends on how you define the UDF. 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* argument 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. So a table

function that never returns the end-of-table condition (SQL-state value '02000') can cause an infinite processing loop if you use it with a GROUP BY or ORDER BY clause.

Table Function Error Processing

The error processing model for table function calls is as follows:

1. If FIRST call fails, no further calls are made.
2. If FIRST call succeeds, the nested OPEN, FETCH, and CLOSE calls are made, and the FINAL call is always made.
3. If OPEN call fails, no FETCH or CLOSE call is made.
4. If OPEN call succeeds, then FETCH and CLOSE calls are made.
5. If a FETCH call fails, no further FETCH calls are made, but the CLOSE call is made.

Note: This model describes the ordinary error processing for scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made. For example, for a FENCED UDF, if the `db2udf` fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

Scalar Function Error Processing

The error processing model for scalar UDFs which are defined with the FINAL CALL specification is as follows:

- If FIRST call fails, no further calls are made.
- If FIRST call succeeds, then further NORMAL calls are made as warranted by the processing of the statement, and a FINAL call is always made.
- If NORMAL call fails, no further NORMAL calls are made, but the FINAL call is made (if you have specified FINAL CALL).

This means that if an error is returned on a FIRST call, the UDF must clean up before returning, because no FINAL call will be made.

The error processing model for table functions is defined in "Table Function Considerations" on page 441 section of this chapter.

Note: This model describes the ordinary error processing for scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made. For example, for a FENCED UDF, if the `db2udf` fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

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 as 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.

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 */
    sqlint32*           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 */
    sqlint32            start,          /* in:  Substring start value (starts at 1) */
    sqlint32            length,         /* in:  Get this many bytes */
    unsigned char*      buffer_p,       /* in:  Read into this buffer */
    sqlint32*           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 */
    sqlint32            length,         /* in:  Length of data to be appended */
    sqlint32*           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 */
);
```

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.

Return codes. Interpret the return code passed back to the UDF by DB2 for each API as follows:

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.

-3 Bad input value was provided to the API. For examples of bad input values specific to each API, see its description below.

other Invalid locator or other error (for example, memory error). The value that is returned for these cases is the SQLCODE corresponding to the error condition. For example, -423 means invalid locator. Please note that before returning to the UDF with one of these "other" codes, DB2 makes a judgment as to the severity of the error. For severe errors, DB2 remembers that the error occurred, and when the UDF returns to DB2, regardless of whether the UDF returns an error SQLSTATE to DB2, DB2 takes action appropriate for the error condition. For non-severe errors, DB2 forgets that the error has occurred, and leaves it up to the UDF to decide whether it can take corrective action, or return an error SQLSTATE to DB2.

- `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.

  A return code of **3** may indicate:

  – udfloc_p (address of locator) is zero

  – return_len_p (address of where to put length) is zero

- `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.

A return code of **3** may indicate:

– udfloc_p (address of locator) is zero
– start is less than 1
– length is negative
– buffer_p (buffer address) is zero
– return_len_p (address of where to put length) is zero

- `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 append 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 effect on any LOB values in the SQL statement, or stored in the database.

  This API can be used to build very large LOB values in a piecemeal manner. In cases where a large number of appends is used to build a result, the performance of this task can be improved by:

  – allocating a large application control heap (APP_CTL_HEAP_SZ is the database manager configuration parameter)
  – doing fewer appends of larger buffers; for example, instead of doing 20 appends of 50 bytes each, doing a single 1000 byte append

  SQL applications which build many large LOB values via the sqludf_append() API may encounter errors caused by limitations on the amount of disk space available. The chance of these errors happening can be reduced by:

  – using larger buffers for the individual appends
  – doing frequent COMMITs between statements

– in cases where each row of a SELECT statement is building a LOB value via this API, using a CURSOR WITH HOLD and doing COMMITs between rows

A return code of **3** may indicate:

– udfloc_p (address of locator) is zero
– length is negative
– buffer_p (buffer address) is zero

- `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.

  A return code of **3** may indicate:

  – udfloc_p (address of locator) is zero
  – loc_type is not one of the three valid values
  – loc_p (address of where to put locator) is zero

- `sqludf_free_locator()`

  Frees the passed locator.

  Use this API to free any locators that were created with the sqludf_create_locator() API, and which were used only for internal manipulation. It is NOT NECESSARY to free locators passed into the UDF. It is NOT NECESSARY to free any locator created by the UDF via sqludf_create_locator() if that locator is passed out of the UDF as an output.

  A return code of **3** may indicate:

  – udfloc_p (address of locator) is zero

The following notes apply to the use of these APIs:

Notes:

1. A UDF which is defined to return a LOB locator has several possibilities available to it. It can return:
   - an input locator passed to it
   - an input locator passed to it which has been appended to via sqludf_append()
   - a locator created to via sqludf_create_locator(), and appended to via sqludf_append()
2. A table function can be defined as returning one or more LOB locators. Each of them can be any of the possibilities discussed in the preceding

item. It is also valid for such a table function to return the same locator as an output for several table function columns.

3. A LOB locator passed to a table function as an input argument remains alive for the entire duration of the row generation process. In fact, the table function can append to a LOB using such a LOB locator while generating one row, and see the appended bytes on a subsequent row.
4. The internal control mechanisms used to represent a LOB which originated in DB2 as a LOB locator output from a UDF (table or scalar function), take 1950 bytes. For this reason, and because there are limitations on the size of a row which is input to a sort, a query which attempts to sort multiple such LOBs which originated as UDF LOB locators will be limited to (at most) two such values per row, depending on the sizes of the other columns involved. The same limitation applies to rows being inserted into a table.

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.

- 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()`.

- 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 1 000 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 1 000 each time through the loop. It proceeds in this manner until it finds what it is looking for.

- 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.

- 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:

- **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.
- 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.
- 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 have no effect on SQL values or processing, but may cause DB2 to malfunction.
- 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 34, (`Failure to initialize OLE library`).
- Re-entrancy 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.
- 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. Your operating system may also prevent you from changing a UDF body that is in use. This practice is not recommended.

- 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.
- 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.
- 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.
- 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.
- Regarding the choice between FENCED and UNFENCED:

  FENCED UDF
  A FENCED UDF runs in its own process and thus cannot access most DB2 internal control and data areas, whether inadvertently or deliberately. This makes a FENCED UDF a safer choice for the database. However, it is still possible for a FENCED UDF that contains programming errors to bring down DB2, though not as easy as for a NOT FENCED UDF. A UDF that performs a massive overwrite of a return variable, for example, can cause DB2 to abend.

  UNFENCED UDF
  A NOT FENCED UDF performs better than a FENCED UDF, because a NOT FENCED UDF is loaded and executed directly in the DB2 engine process. NOT FENCED UDFs avoid the performance expense of process communication overhead. However, a NOT FENCED UDF could conceivably access or alter DB2 internal control or data areas. It is easier for an improperly written NOT FENCED UDF to bring down DB2 than a FENCED UDF.

  Obviously, with both FENCED and NOT FENCED UDFs, you should:
  – ensure the UDF is robustly written
  – subject the UDF to a rigorous design and code review

– test the UDF in an environment where no harm can be done if it is not correctly written; for example, a test database.

Most abends caused by a UDF are caught by DB2, which returns a -430 SQLCODE and prevents the database from being corrupted. However, certain types of UDF misbehavior, including a massive overwrite of a return value buffer, can cause DB2 to fail as well as the UDF. Pay attention particularly to any UDF which returns variable-length data, or which calculates how many bytes it must move to the return value buffer.

- For considerations on using UDFs with EUC code sets, see "Considerations for UDFs" on page 525.
- For an application running NOT FENCED 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 NOT FENCED 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 NOT FENCED and FENCED UDFs, two separate blocks of memory, each of the size indicated by the `UDF_MEM_SZ` parameter are used. Refer to the *Administration Guide* for more information about this configuration parameter.
- 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 439.)
  – 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.

      Other system calls may also cause problems if they interfere with the normal operation of DB2; for example, a UDF that attempts to unload a library containing a UDF from memory could cause severe problems. Be careful in coding and testing any UDFs containing system calls.

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, refer to 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 user ID of the DB2 Agent Process (NOT FENCED), or the user ID which owns the `db2udf` executable (FENCED). This user ID controls the system resources available to the UDF. For information on the `db2udf` executable, refer to the *Quick Beginnings* for your platform.
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 "Appendix B. Sample Programs" on page 743.

For information on compiling and linking UDFs, refer to the *Application Building Guide*.

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 populating the table, issue the following statement 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. You receive 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.
```

Refer to the previous information on table TEST 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. (Note that if you enable *friendly arithmetic* with the DFT_SQLMATHWARN configuration parameter, DB2 returns a NULL instead of an error in this situation.) 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>

 /************************************************************************
*  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
************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN divid (
   sqlint32 *num,                       /* numerator */
   sqlint32 *denom,                     /* denominator */
   sqlint32 *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 */
}
/* end of UDF : divid */
```

For this UDF, notice that:

- 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 (if you do not enable *friendly arithmetic* with the database configuration parameter DFT_SQLMATHWARN):

```
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.

While the preceding example does not consider the BIGINT data type, you can easily extend the example to include BIGINT.

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
*****************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
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( ( char * ) 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( ( char * ) out->data, &in1->data[*in2], len1 - *in2 ) ;
      strncpy( ( char * ) &out->data[len1 - *in2], in1->data, *in2 ) ;
      out->length = in1->length;
   } /* endif */
   *outnull = 0;                       /* result is always non-NULL */
}
/* end of UDF : fold */

 /*****************************************************************************
*  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
*****************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
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( ( char * ) in )) {              /* no vowels found */
         /* error state */
      strcpy( ( char * ) sqludf_sqlstate, "38999" ) ;
         /* message insert */
      strcpy( ( char * ) sqludf_msgtext, "findvwl: No Vowel" ) ;
   } else {                              /* a vowel was found at "i" */
      *out = i + 1;
      *outnull = 0;
   } /* endif */
}
/* 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>

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
   sqlint32 len;
   sqlint32 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
*************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN ctr (
   sqlint32 *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;
}
/* end of UDF : ctr */
```

For this UDF, observe that:

- It includes `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 <sql.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#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 descriptor structure */
typedef struct {
  char fld_field[31] ;                    /* Field data */
  int  fld_ind ;            /* Field null indicator data */
  int  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, SQL_TYP_VARCHAR, 30,  0 }, /* city         */
```

```
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  2 }, /* temp_in_f     */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  7 }, /* humidity      */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR,  5, 13 }, /* wind          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3, 19 }, /* wind_velocity */
  { "", SQL_ISNULL, SQL_TYP_FLOAT,    5, 24 }, /* barometer     */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast      */
  /* You may want to add more fields here */

  /* Do not forget a null termination */
  { ( char ) 0, 0, 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",
   .
   .
   .
   "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
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* 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 ) ;

}
```

```
#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* 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 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* 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 SQL_TYP_VARCHAR:
                 strcpy( field->fld_field,
                         ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_INTEGER:
                 int_ptr = ( int * ) field->fld_field ;
```

```
                *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                break ;
           case SQL_TYP_FLOAT:
                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 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN 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 ;

  /* 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' ;

     /* Clean all field data and field null indicator data */
     clean_fields( 0 ) ;

     /* Fills city field null indicator data */
     fields[0].fld_ind = SQL_NOTNULL ;

     /* Find a full city name using a short name */
     /* Fills city field data */
     if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
        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
  NO 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.
- NO FINAL CALL is specified.

- 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 */
          sqlint16 * inp_nul,
          sqlint16 * cri_nul,
          sqlint16 * 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 */
       sqlint32  input_len;  /* receiver for input LOB length */
       sqlint32  input_pos;  /* current position for scanning input LOB */
       char lob_buf[100];    /* data buffer */
       sqlint32  input_rec;  /* number of bytes read by sqludf_substr */
       sqlint32  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.
- The UDF does not test whether either input argument is null, 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 `CCounter` 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 `CCounter` 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 `CCounter` 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;
```

While processing the following query, DB2 creates two different instances of class `CCounter`. An instance is created 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.

```
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.
```

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 true for both FENCED and NOT FENCED UDFs, as both types can cause DB2 to malfunction if they are incorrectly written. Defining a UDF as FENCED offers more protection against integrity and security exposures than NOT FENCED, but there is no guarantee. Good coding practices, including reviews and testing, should prevail in either case.

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 -450 (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 with SQLCODE -431 (SQLSTATE 38504).

A massive overwrite of a return value buffer, even in a FENCED UDF, can cause both the UDF and DB2 to abnormally terminate. Pay special attention when designing, coding, and reviewing any UDF that moves bytes to return value buffers. Be careful with any UDF which, for example, calculates how many bytes to move before moving the bytes. In C, memcpy is often used for this function. Closely examine the boundary cases (extra short and long values) for UDFs that move bytes to a return value buffer.

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 user ID 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 16. Using Triggers in an Active DBMS

Why Use Triggers?

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.

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 (refer to 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 of Triggers

Using triggers in your database manager can result in:

- **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.

- **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.

- **Easier maintenance**

  If a business policy changes, only the corresponding trigger needs to change instead of each application program.

Overview of a Trigger

When you create a trigger, you associate it with a table. This table is called the *subject table* of the trigger. The term *update operation* refers to any change in the state of the subject table. An update operation is initiated by:

- an INSERT statement
- an UPDATE statement, or a referential constraint which performs an UPDATE
- a DELETE statement, or a referential constraint which performs a DELETE

You must associate each trigger with one of these three types of update operations. The association is called the *trigger event* for that particular trigger.

You must also define the action, called the *triggered action*, that the trigger performs when its trigger event occurs. The triggered action consists of one or more SQL statements which can execute either before or after the database manager performs the trigger event. Once a trigger event occurs, the database manager determines the set of rows in the subject table that the update operation affects and executes the trigger.

When you create a trigger, you declare the following attributes and behavior:

- The name of the trigger.
- The name of the subject table.
- The trigger activation time (BEFORE or AFTER the update operation executes).
- 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.

For more information on the CREATE TRIGGER statement, refer to the *SQL Reference*.

Trigger Event

Every trigger is associated with an event. Triggers are activated when their corresponding event occurs in the database. This trigger event occurs when the specified action, 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 the trigger `new_hire`, which activates when you perform an insert operation on table `employee`.

You associate every trigger event, and consequently every trigger, with exactly one subject table and exactly one update operation. The update 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 trigger event 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`, activates only if you perform an update operation 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 trigger event defines a set of rows in the subject table that is affected by that SQL operation. For example, suppose you run the following UPDATE statement on the `parts` table:

```
UPDATE PARTS
  SET ON_HAND = ON_HAND + 100
  WHERE  PART_NO > 15000
```

The set of affected rows for the associated 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 entire trigger event.

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" on page 488).

Trigger Activation Time

The *trigger activation time* specifies when the trigger should be activated. That is, either BEFORE or AFTER the trigger event executes. 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 trigger event executes. 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 trigger event executes, and after the database manager checks all constraints that the trigger event may affect, 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 trigger event is applied to the database. Consequently, they are activated before integrity constraints are checked and may be violated by the trigger event.

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 triggered action. The set of SQL operations you can specify in BEFORE and AFTER triggers are described in "Triggered Action" on page 492. 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 triggered action, 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 triggered action 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 triggered action 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 triggered action 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 triggered action 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 triggered action 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 action 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, refer to 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, refer to the *SQL Reference*.

UDFs are written in either the SQL, Java, C, or C++ programming language. This enables complex control of logic flows, error handling and recovery, and access to system and library functions. (See "Chapter 15. Writing User-Defined Functions (UDFs) and Methods" on page 393 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, refer to 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 trigger event 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 trigger event and referential constraint actions caused directly or indirectly (that is, recursively by other referential constraints) by the triggered action. 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

The following section describes how to exploit triggers and constraints to model application structures that use UDTs, UDFs, and LOBs. 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

You could write an application that stores complete electronic mail messages as a LOB value within the column MESSAGE of the ELECTRONIC_MAIL table. To manipulate the electronic mail, you could use UDFs 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
```

Part 5. DB2 Programming Considerations

Chapter 17. Programming in Complex Environments

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
- Conversion Between Different Code Pages
- Deriving Code Page Values
- Deriving Locales in Application Programs

- National Language Support Application Development

Collating Sequence Overview

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, and 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.

Each single-byte character in a database 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; the assignment of numbers to characters in a set is collectively called a *code page*. 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. In the simplest collating sequence, the weights are identical to the code points. This is called the *identity sequence*.

For example, suppose the characters B and b have the code points X'42' and X'62', respectively. If (according to the collating sequence table) they both have a sort weight of X'42' (B), they collate the same. If the sort weight for B is X'9E', and the sort weight for b is X'9D', b will be sorted before B. The collation sequence table specifies the weight of each character. The table is different from a code page, which specifies the code point of each character.

Consider the following example. The ASCII characters A through Z are represented by X'41' through X'5A'. To describe a collating sequence in which these characters are sorted consecutively (no intervening characters), you can write: X'41', X'42', ... X'59', X'5A'.

The hexadecimal value of a multi-byte character is also used as the weight. For example, suppose the code points for the double-byte characters A and B are X'8260' and X'8261' respectively, then the collation weights for X'82', X'60', and X'61' are used to sort these two characters according to their code points.

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 a collating sequence can be simplified if the collating sequence provides weights for all 256 code points. The weight of each character can be determined using the code point of the character.

In all cases, DB2 uses the collation table that was specified at database creation time. If you want the multi-byte characters to be sorted the way that they appear in their code point table, you must specify IDENTITY as the collation sequence when you create the database.

Note: For double-byte and Unicode characters in GRAPHIC fields, the sort sequence is always IDENTITY.

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 can become a two-phase process:

1. Compare the characters in each string based on their weights.
2. If step 1 yields 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 about character comparisons, refer to the *SQL Reference*.

Case Independent Comparisons: To perform character comparisons that are independent of case, you can use the TRANSLATE function to select and compare mixed case column data by translating it to uppercase (for purposes of comparison only). Consider the following data:

```
Abel
abels
ABEL
abel
ab
Ab
```

The following SELECT statement:

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

returns

```
ab
Ab
abel
Abel
ABEL
abels
```

You could also specify the following SELECT statement when creating view ″v1″, make all comparisons against the view in uppercase, and request table INSERTs 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 `sqlecrea` - 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. ″A″ will always come before ″a″, because they are equal in every sense. The only basis upon which to sort is the hexadecimal value.

Thus

```
SELECT c1 FROM T1 WHERE c1 LIKE 'ab%'
```

returns

```
ab
abel
abels
```

and

```
SELECT c1 FROM T1 WHERE c1 LIKE 'A%'
```

returns

```
Abel
Ab
ABEL
```

The following statement

```
SELECT c1 FROM T1 ORDER BY c1
```

returns

```
ab
Ab
abel
Abel
ABEL
abels
```

Thus, you may want to consider using the scalar function TRANSLATE(), as well as `sqlecrea`. Note that you can only specify a collating sequence using `sqlecrea`. You cannot specify a collating sequence from the command line processor (CLP). For information about the TRANSLATE() function, refer to the *SQL Reference*. For information about `sqlecrea`, refer to the *Administrative 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'
```

Collating Sequence Sort Order: EBCDIC and ASCII Example

The order in which data in a database is sorted depends on the collating sequence defined for the database. For example, suppose that database A uses the EBCDIC code page's default collating sequence and that database B uses the ASCII code page's default collating sequence. Sort orders at these two databases would differ, as shown in Figure 19.

```
SELECT.....
  ORDER BY COL2

EBCDIC-Based Sort        ASCII-Based Sort

COL2                     COL2
----                     ----
V1G                      7AB
Y2W                      V1G
7AB                      Y2W
```

Figure 19. Example of How a Sort Order in an EBCDIC-Based Sequence Differs from a Sort Order in an ASCII-Based Sequence

Similarly, character comparisons in a database depend on the collating sequence defined for that database. So if database A uses the EBCDIC code page's default collating sequence and database B uses the ASCII code page's default collating sequence, the results of character comparisons at the two databases would differ. Figure 20 on page 508 illustrates the difference.

```
SELECT.....
  WHERE COL2 > 'TT3'

EBCDIC-Based Results     ASCII-Based Results

COL2                     COL2
----                     ----
TW4                      TW4
X72                      X72
39G
```

Figure 20. Example of How a Comparison of Characters in an EBCDIC-Based Sequence Differs from a Comparison of Characters in an ASCII-Based Sequence

If you are creating a federated database, consider specifying that your collating sequence matches the collating sequence at a data source. This approach will maximize "pushdown" opportunities and possibly increase query performance. For more information on the relationship between pushdown analysis, collating sequences, and query performance, refer to the *Administration Guide: Implementation*.

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/region 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/region 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 code point 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 for C and C++" on page 595
- For COBOL, "Include Files for COBOL" on page 680
- For FORTRAN, "Include Files for FORTRAN" on page 702.

Deriving Code Page Values

The *application code page* is derived from the active environment when the database connection is made. If the DB2CODEPAGE registry variable is set, its value is taken as the application code page. However, it is not necessary to set the DB2CODEPAGE registry variable because DB2 will determine the appropriate code page value from the operating system. Setting the DB2CODEPAGE registry variable to incorrect values may cause unpredictable results.

The *database code page* is derived from the value specified (explicitly or by default) at the time the database is created. For example, the following defines how the *active environment* is determined in different operating environments:

UNIX On UNIX based operating systems, the active environment is determined from the locale setting, which includes information about language, territory and code set.

OS/2 On 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.

Windows 32-bit operating systems

For all Windows 32-bit operating systems, if the DB2CODEPAGE environment variable is not set, the code page is derived from the ANSI code page setting in the Registry.

For a complete list of environment mappings for code page values, refer to the *Administration Guide*.

Deriving Locales in Application Programs

Locales are implemented one way on Windows and another way on UNIX based systems. There are two locales on UNIX based systems:

- 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 running.

On Windows, cultural preferences can be set through Regional Settings on the Control Panel. However, there is no environment locale like the one on UNIX based systems.

When your program is started, it gets a default C locale. It does *not* get a copy of the environment locale. If you set the program locale to any locale other than "C", DB2 Universal Database uses your current program locale to determine the code page and territory settings for your application environment. Otherwise, these values are obtained from the operating system environment. Note that `setlocale()` is not thread-safe, and if you issue `setlocale()` from within your application, the new locale is set for the entire process.

How DB2 Derives Locales

On UNIX based systems, 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 "C", DB2 will use this value to determine the application code page by mapping it to its corresponding code page.
- If LC_CTYPE has a 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 assume the default of the US English environment, and code page 819 (ISO 8859-1).

- If LC_CTYPE no longer has a value of "C", its new value will be used to map to a corresponding code page. For information about the default locale for a particular platform, refer to the *Administration Guide*. For additional information on building applications on a particular platform, refer to the *Application Building Guide*.

National Language Support Application Development

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.

If your program contains constant character strings, it is strongly recommended that you precompile, bind, compile, and execute the application using the same code page. For a Unicode database, you should use host variables instead of using string constants. This is because data conversions by the server can occur in both the bind and the execution phases. This could be a concern if constant character strings are used within the program. These embedded strings are converted at bind time based on the code page which is in effect during the bind phase. Seven-bit ASCII characters are common to all the code pages supported by DB2 Universal Database and will not cause a problem. For non-ASCII characters, 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 509.

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 in C and C++" on page 621. If you deal with EUC code sets in your applications, refer to "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 521 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 must be

characters supported by your application code page. For more information about extended character sets, refer to the *SQL Reference*.

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 code page 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 19. 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'
1363	X'A3A5'	X'A3DF'	X'A1A1'	X'A1E0'
1386	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
5039	X'8193'	X'8151'	X'8140'	X'FCFC'

Unicode Considerations: For Unicode databases, the GRAPHIC space is X'0020', which is different from the GRAPHIC space of X'3000' used for euc-Japan and euc-Taiwan databases. Both X'0020' and X'3000' are space characters in the Unicode standard. The difference in the GRAPHIC space code points should be taken into consideration when comparing data from these EUC databases to data from Unicode database.

For more information about Unicode databases, see the *Administration Guide: Planning*.

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/region 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 eight 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, 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 code page 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 to maintain data integrity.

When Does Code Page Conversion Occur?: Code page 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. For a list of supported code page conversions, refer to the *Administration Guide*.

- 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 host or AS/400 server. In this case the data receiver converts the character data. 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 User's Guide*.

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.[1] See the *SQL Reference* for unequal code page rules for assigning, comparing, and combining strings.
- 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 Code Page 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 Code Page 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.

1. 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.

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. To avoid a possible loss of character data, ensure you follow the supported character conversions listed in the *Administration Guide*. 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 not necessarily provide meaningless conversions in the two-step conversion process mentioned above). Such problems are avoided if the application code page is the same as the database code page.
- 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. For a summary of these rules, and for specific application with individual operators and predicates, refer to the *SQL Reference*.

For a list of the code pages supported by DB2 Universal Database, refer to the *Administration Guide*. The values under the heading "Group" can be used to determine where conversions 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: Code page 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.

Code Page 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 526 for information on dealing with unequal code page situations.

DBCS Character Sets

Each combined single-byte character set (SBCS) or double-byte character set (DBCS) code page allows for both single- and double-byte character code points. This is usually accomplished by reserving a subset of the 256 available code points of a mixed code table for single-byte characters, with the remainder of the code points either undefined, or allocated to the first byte of double-byte code points. These code points are shown in the following table.

Table 20. Mixed Character Set Code Points

Country/Region	Supported Mixed Code Page	Code Points for Single-byte Characters	Code Points for First Byte of Double-byte Characters
Japan	932, 943	X'00'-X'7F', X'A1'-X'DF'	X'81'-X'9F', X'E0'-X'FC'
Japan	942	X'00'-X'80', X'A0'-X'DF', X'FD'-X'FF'	X'81'-X'9F', X'E0'-X'FC'
Taiwan	938 (*)	X'00'-X'7E'	X'81'-X'FC'
Taiwan	948 (*)	X'00'-X'80', X'FD', X'FE'	X'81'-X'FC'
Korea	949	X'00'-X'7F'	X'8F'-X'FE'

Table 20. Mixed Character Set Code Points (continued)

Country/Region	Supported Mixed Code Page	Code Points for Single-byte Characters	Code Points for First Byte of Double-byte Characters
Taiwan	950	X'00'-X'7E'	X'81'-X'FE'
China	1381	X'00'-X'7F'	X'8C'-X'FE'
Korea	1363	X'00'-X'7F'	X'81'-X'FE'
China	1386	X'00'	X'81'-X'FE'
Note: (*) This is an old code page that is no longer recommended.			

Code points not assigned to either of these categories are not defined, and are processed as single-byte undefined code points.

Within each implied DBCS code table, there are 256 code points available as the second byte for each valid first byte. Second byte values can have any value from X'40' to X'7E', and from X'80' to X'FE'. Note that in DBCS environments, DB2 does not perform validity checking on individual double-byte characters.

Extended UNIX Code (EUC) Character Sets

Each EUC code page allows for both single-byte character code points, and up to three different sets of multi-byte character code points. This is accomplished by reserving a subset of the 256 available code points of each implied SBCS code page identifier for single-byte characters. The remainder of the code points is undefined, allocated as an element of a multi-byte character, or allocated as a single-shift introducer of a multi-byte character. These code points are shown in the following tables.

Table 21. Japanese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	n/a	n/a
G3	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	n/a

Table 22. Korean EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a

Table 22. Korean EUC Code Points (continued)

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

Table 23. Traditional Chinese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	X'A1'-X'FE'
G3	n/a	n/a	n/a	n/a

Table 24. Simplified Chinese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

Code points not assigned to any of these categories are not defined, and are processed as single-byte undefined code points.

Running CLI/ODBC/JDBC/SQLJ Programs in a DBCS Environment

JDBC and SQLJ programs access DB2 using the DB2 CLI/ODBC driver and therefore use the same configuration file (db2cli.ini). The following entries must be added to this configuration file if you run Java programs that access DB2 Universal Database in a DBCS environment:

PATCH1 = 65536

This forces the driver to manually insert a ″G″ in front of character literals which are in fact graphic literals. This PATCH1 value should always be set when working in a double byte environment.

PATCH1 = 64

This forces the driver to NULL terminate graphic output strings. This is needed by Microsoft Access in a double byte environment. If you need to use this PATCH1 value as well then you would add the two values together (64+65536 = 65600) and set PATCH1=65600. See Note #2 below for more information about specifying multiple PATCH1 values.

PATCH2 = 7

This forces the driver to map all graphic column data types to char column data type. This is needed in a double byte environment.

PATCH2 = 10

This setting should only be used in an EUC (Extended Unix Code) environment. It ensures that the CLI driver provides data for character variables (CHAR, VARCHAR, etc...) in the proper format for the JDBC driver. The data in these character types will not be usable in JDBC without this setting.

Note:

1. Each of these keywords is set in each database specific stanza of the db2cli.ini file. If you want to set them for multiple databases then you need to repeat them for each database stanza in db2cli.ini.
2. To set multiple PATCH1 values you add the individual values and use the sum. To set PATCH1 to both 64 and 65536 you would set PATCH1=65600 (64+65536). If you already have other PATCH1 values set then replace the existing number with the sum of the existing number and the new PATCH1 values you want to add.
3. To set multiple PATCH2 values you specify them in a comma delimited string (unlike the PATCH1 option). To set PATCH2 values 1 and 7 you would set PATCH2="1,7"

For more information about setting these keywords refer to the *Installation and Configuration Supplement*.

Japanese and Traditional Chinese EUC and UCS-2 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 Unicode UCS-2 code set, a double-byte encoding scheme which is a proper subset of the full Unicode character repertoire. 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 is 1200, and the CCSID number for the same code page is 13488. Graphic data in an eucJP or eucTW database uses the CCSID number 13488. In a Unicode database, use CCSID 1200 for GRAPHIC data.

DB2 Universal Database supports the all the Unicode characters that can be encoded using UCS-2, but does not perform any composition, decomposition, or normalization of characters. More information about the Unicode standard can be found at the Unicode Consortium web site, `www.unicode.org`, and from the latest edition of the Unicode Standard book published by Addison Wesley Longman, Inc.

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 "Code Page Conversion Expansion Factor" on page 517. 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 128 bytes. It is possible that a character string which is 128 bytes under a double-byte code page is larger, say 135 bytes, under an EUC code page. This hypothetical 135-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. Administering an EUC database from a double-byte client imposes fewer restrictions then administering a double-byte database from an EUC client. Double-byte character strings typically are equal or shorter than the corresponding EUC character string. This will generally lead to less problems caused by enforcing the character string length limits.

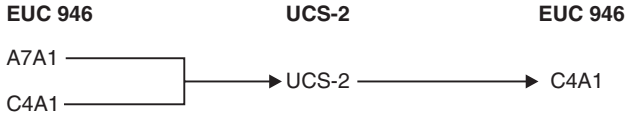
Note: In the case of SQL statements, validation of input parameters is not conducted until the entire statement has been converted to the database code page. Thus you can use character strings which may be technically longer then allowed when they represented in the client code page, but which meet length 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:



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 online Product and Service Technical Library (http://www.ibm.com/software/data/db2/library/).

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, see "Considerations for Stored Procedures" on page 525. Additional language-specific application development issues are discussed in:

- "Japanese or Traditional Chinese EUC, and UCS-2 Considerations in C and C++" on page 626 (for C and C++).
- "Japanese or Traditional Chinese EUC, and UCS-2 Considerations for COBOL" on page 699 (for COBOL).
- "Japanese or Traditional Chinese EUC, and UCS-2 Considerations for FORTRAN" on page 715 (for FORTRAN).
- "Japanese or Traditional Chinese EUC Considerations for REXX" on page 734 (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, refer to 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, refer to 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/region, 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

- Code Page Conversion String Length Overflow
- Applications Connected to a Unicode Database
- Rules for String Conversions
- Character Conversions Past Data Type Limits
- Code Page 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	32 672
LONG VARCHAR	32 700
CLOB	2 147 483 647

4. Determine the maximum target length.

```
1  if temp_target_length < actual_source_length
      max_target_length = type_maximum_length
   else
2  if temp_target_length > type_maximum_length
      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 533 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 25 lists values to expect for various application code page and database code page combinations when using DB2 Universal Database.

Table 25. 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

Table 25. SQLCA.SQLERRD Settings on CONNECT (continued)

Application Code Page	Database Code Page	SQLERRD(1)	SQLERRD(2)
eucKR	eucKR	+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 128 bytes in length. It is possible that a character string which is 128 bytes in length encoded under a DBCS code page expands past the 128 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(128) column is retrieved. Under circumstances where the database and application code pages are equal, the length of the data returned is 128 bytes. However, in an unequal code page situation 128 bytes of data encoded under a DBCS code page may expand past 128 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 code page 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 a table provided by an end-user, and checks that the table name is not greater than 128 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 128 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 131 bytes in length. Since there is no expansion for DBCS to DBCS connections, the table name is 128 bytes in the database environment, and the CREATE TABLE is successful.
2. An EUC client connects to the DBCS database. It creates a table (t2) with a table name which is 120 bytes long when encoded as EUC and 100 bytes long when converted to DBCS. The table name in the DBCS database is 100 bytes. The CREATE TABLE is successful.
3. The EUC client creates a table (t3) with a table name that is 64 EUC characters in length (131 bytes). When this name is converted to DBCS its length shrinks to the 128 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	The application considers the table name invalid because it is 131 bytes long.
t2	Displays correct results
t3	The application considers the table name invalid because it is 131 bytes long.

5. The EUC client is used to query the DBCS database from the CLP. Although the table name is 131 bytes long on the client, the queries are successful because the table name is 128 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 "Code Page Conversion Expansion Factor" on page 517 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, see "Deriving Code Page Values" on page 509, 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 531).

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.

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.

Code Page 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 128 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, refer to 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 32672 bytes. In cases where expansion does exceed the maximum length of the data type, an SQLCODE -334 (SQLSTATE 22524) is returned.

Code Page 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.

Applications Connected to a Unicode Database

Note that the information contained in the previous section, "Developing for Mixed Code Set Environments" on page 525, is also applicable to a Unicode database.

Applications from any code page environment can connect to a Unicode database. For applications that connect to a Unicode database, the database manager converts character string data between the application code page and the database code page (UTF-8). For a Unicode database, GRAPHIC data is in UCS-2 big-endian order. However, when you use the command line processor to retrieve graphic data, the graphic characters are also converted to the client code page. This conversion allows the command line processor to display graphic characters in the current font. Data loss may occur whenever the database manager converts UCS-2 characters to a client code page. Characters that the database manager cannot convert to a valid character in the client code page are replaced with the default substitution character in that code page.

When DB2 converts characters from a code page to UTF-8, the total number of bytes that represent the characters may expand or shrink, depending on the code page and the code points of the characters. 7-bit ASCII remains invariant in UTF-8, and each ASCII character requires one byte. Non-ASCII characters become more than one byte each. For more information about UTF-8 conversions, refer to the *Administration Guide*, or refer to the Unicode standard documents.

For applications that connect to a Unicode database, GRAPHIC data is already in Unicode. For applications that connect to DBCS databases, GRAPHIC data is converted between the application DBCS code page and the database DBCS code page. Unicode applications should perform the necessary conversions to and from Unicode themselves, or should set WCHARTYPE CONVERT option and use `wchar_t` for graphic data. For more details about this option, please see "Handling Graphic Host Variables in C and C++" on page 621.

Considerations for Multisite Updates

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
- Multisite Update

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, refer to 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.

Multisite Update

Multisite update, also known as Distributed Unit of Work (DUOW) and Two-Phase commit, is a function that enables your applications to update data in multiple remote database servers with guaranteed integrity. A good example of a multisite update is a banking transaction that involves transfer of money from one account to another in a different database server. In such a transaction it is critical that updates that implement debit operation on one account do not get committed unless updates required to process credit to the other account are committed as well. The multisite update considerations apply when data representing these accounts is managed by two different database servers.

You can use multisite update to read and update multiple DB2 Universal Database databases within a unit of work. If you have installed DB2 Connect or use the DB2 Connect capability provided with DB2 Universal Database Enterprise Edition you can also use multisite update with host or AS/400 database servers, such as DB2 Universal Database for OS/390 and DB2

Universal Database for AS/400. Certain restrictions apply when you use multisite update with other database servers, as described in "Multisite Update with DB2 Connect" on page 799.

A transaction manager coordinates the commit among multiple databases. If you use a transaction processing (TP) monitor environment such as TxSeries CICS, the TP monitor uses its own transaction manager. Otherwise, the transaction manager supplied with DB2 is used. DB2 Universal Database for OS/2, UNIX, and Windows 32-bit operating systems is an XA (extended architecture) compliant resource manager. Host and AS/400 database servers that you access with DB2 Connect are XA compliant resource managers. 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 multisite update, refer to the *Administration Guide*.

When to Use Multisite Update

Multisite Update 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 Multisite Update Application

Table 26 on page 537 illustrates how you code SQL statements for multisite update. The left column shows SQL statements that do not use multisite update; the right column shows similar statements with multisite update.

Table 26. RUOW and Multisite Update SQL Statements

RUOW Statements	Multisite Update Statements

CONNECT TO D1
  SELECT
  UPDATE
  COMMIT
 	
CONNECT TO D2
  INSERT
  COMMIT
 	
CONNECT TO D1
  SELECT
  COMMIT
CONNECT RESET
	CONNECT TO D1
	  SELECT
	  UPDATE
	
	CONNECT TO D2
	  INSERT
	  RELEASE CURRENT
	
	SET CONNECTION D1
	  SELECT
	  RELEASE D1
	  COMMIT

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 multisite update application.

Some SQL statements are coded and interpreted differently in a multisite update 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 host or AS/400 database servers that you access with DB2 Connect without starting the DB2 Connect sync point manager.
  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 Multisite Update Application

When you precompile a multisite update 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 more information on setting the connection type, refer to the *Command Reference*. The following precompiler options are used when you precompile an application which uses multisite updates:

CONNECT (1 | 2)
Specify CONNECT 2 to indicate that this application uses the SQL syntax for multisite update applications, as described in "Coding SQL for a Multisite Update Application" on page 536. The default, CONNECT 1, means that the normal (RUOW) rules for SQL syntax apply to the application.

SYNCPOINT (ONEPHASE | TWOPHASE | NONE)
If you specify SYNCPOINT TWOPHASE and DB2 coordinates the transaction, DB2 requires a database to maintain the transaction state information. When you deploy your application, you must define this database by configuring the database manager configuration parameter TM_DATABASE. For more information on the TM_DATABASE database manager configuration parameter, refer to the *Administration Guide*. 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/ANSI SQL92 should be used in multisite update 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, refer to the *Command Reference*.

Multisite update 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 multisite update 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 Multisite Update Application

For information on performing multisite updates coordinated by an XA transaction manager with connections to a host or AS/400 database, refer to the *DB2 Connect User's Guide*.

The following configuration parameters affect applications which perform multisite updates. With the exception of LOCKTIMEOUT, the configuration parameters are database manager configuration parameters. LOCKTIMEOUT is a database configuration parameter.

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 for a given database. The application must issue an explicit ROLLBACK to roll back all databases that participate in the multisite update. LOCKTIMEOUT is a database configuration parameter.

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 host or AS/400 server using SNA.

SPM_NAME

- If SPM is being used with a TCP/IP 2PC connection then the SPM_NAME must be an unique identifier within the network. When you create a DB2 instance, DB2 derives the default value of SPM_NAME from the TCP/IP hostname. You may modify this value if it is not acceptable in your environment. For TCP/IP

connectivity with host database servers, the default value should be acceptable. For SNA connections to host or AS/400 database servers, this value must match an SNA LU profile defined within your SNA product.

- If SPM is being used with an SNA 2PC connection, the SPM name must be set to the LU_NAME that is used for 2PC.
- If SPM is being used for both TCP/IP and SNA then the LU_NAME that is used for 2PC must be used.

Note: Multisite updates in an environment with host or AS/400 database servers may require SPM. For more information, refer to the *DB2 Connect User's Guide*.

SPM_LOG_SIZE
The number of 4 kilobyte 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, refer to the *Administration Guide*.

Multisite Update Restrictions

The following restrictions apply to multisite update in DB2:

- In a transaction processing (TP) Monitor environment such as TxSeries CICS, 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 multisite update, 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 multisite update 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 multisite update (CONNECT Type 2), as these APIs are not supported in a multisite update:

```
BACKUP DATABASE
BIND
EXPORT
IMPORT
LOAD
MIGRATE DATABASE
PRECOMPILE PROGRAM
RESTART DATABASE
RESTORE DATABASE
REORGANIZE TABLE
ROLLFORWARD DATABASE
```

- Stored procedures are supported within a multisite update. However, a stored procedure that issues a COMMIT and ROLLBACK statement in a multisite update (CONNECT Type 2) receives an SQLCODE -30090 (SQLSTATE 25000) as these statements are not supported in a multisite update.

Accessing Host or AS/400 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, refer to the *SQL Reference* **before** you start coding.
2. Where possible, have your applications check the SQLSTATE rather than the SQLCODE.

   If your applications will use DB2 Connect and you want to use SQLCODEs, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.
3. Test your application with the host or AS/400 databases (such as DB2 Universal Database for OS/390, OS/400, or DB2 for VSE & VM) that you intend to support. For more information, refer to the *DB2 Connect User's Guide*.

For more information on accessing host or AS/400 database systems, see "Appendix D. Programming in a Distributed Environment Programming in a Host or AS/400 Environment" on page 787.

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. To determine if your platform supports the multithreading feature, refer to the *Application Building Guide*.

This section assumes that you are familiar with the terms relating to the development of multithreaded 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 *Administrative API Reference* and "Concurrent Transactions" on page 547.

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.

Multithreaded UNIX Applications Working with Code Page and Country/Region Code

On AIX, Solaris Operating Environment, HP-UX, and Silicon Graphics IRIX, changes have been made to the functions that are used for run time querying of the code page and country/region code to be used for a database connection. They are now thread safe but can create some lock contention (and resulting performance degradation) in a multithreaded application which uses a large number of concurrent database connections.

A new environment variable has been created (DB2_FORCE_NLS_CACHE) to eliminate the chance of lock contention in multithreaded applications. When

DB2_FORCE_NLS_CACHE is set to TRUE the code page and country/region code information is saved the first time a thread accesses it. From that point on the cached information will be used for any other thread that requests this information. By saving this information, lock contention is eliminated and in certain situations a performance benefit will be realized.

DB2_FORCE_NLS_CACHE should not be set to true if the application changes locale settings between connections. If this is done then the original locale information will be returned even after the locale settings have been changed. In general, multithreaded applications will not change locale settings. This ensures that the application remains thread safe.

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 543 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. For information about the concepts and implementation of the XA interface support provided by the database manager, refer to the *Administration Guide: Planning*. To determine if your platform supports the X/Open XA Interface, refer to the *Application Building Guide*.

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 a database 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 a host or AS/400 database that is accessed using SNA connectivity. Local transactions to DB2 for OS/390 Version 5 using TCP/IP connectivity are supported.
- 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 user ID 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 sync point 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 sync point 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. For example, accessing host or AS/400 database servers in environments in which level 2 of DRDA protocol (DRDA2) is not supportable (see "Multisite Update with DB2 Connect" on page 799).
- 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 usable 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 Universal Database for OS/390.
- 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). For a list of these APIs, see "Multisite Update Restrictions" on page 541. These APIs are not supported in a multisite update (Connect Type 2).

- Applications should be single-threaded.

  If you intend to develop a multithreaded application, you should ensure that only one thread uses SQL, or use a multiprocess 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 multithreading, you should configure it to serialize the threads so that one thread will execute to a sync point before another one begins. An example is the **XASerialize** option of ***all_operation*** in AIX/CICS. For more details about the AIX/CICS XAD file which contains this information, refer to the *Administration Guide: Planning*.

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.

Working with Large Volumes of Data Across a Network

You can combine the techniques of stored procedures, described in "Chapter 7. Stored Procedures" on page 193, and row blocking, described in the *Administration Guide: Implementation*, 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 77 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 515 for more information about how and when data conversion occurs.

Chapter 18. Programming Considerations in a Partitioned Environment

Improving Performance

To take advantage of the performance benefits that partitioned environments offer, you should consider using special programming techniques. For example, if your application accesses DB2 data from more than one database manager partition, you need to consider the information contained herein. For an overview of partitioned environments, refer to the *Administration Guide* and the *SQL Reference*.

Using FOR READ ONLY Cursors

If you declare a cursor from which you intend only to read, include FOR READ ONLY or FOR FETCH only in the OPEN CURSOR declaration. (FOR READ ONLY and FOR FETCH ONLY are equivalent statements.) FOR READ ONLY cursors allow the coordinator partition to retrieve multiple rows at a time, dramatically improving the performance of subsequent FETCH statements. When you do not explicitly declare cursors FOR READ ONLY, the coordinator partition treats them as updatable cursors. Updatable cursors incur considerable expense because they require the coordinator partition to retrieve only a single row per FETCH.

Using Directed DSS and Local Bypass

To optimize Online Transaction Processing (OLTP) applications, you may want to avoid simple SQL statements that require processing on all data partitions. You should design the application so that SQL statements can retrieve data from single partitions. These techniques avoid the expense the coordinator partition incurs communicating with one or all of the associated partitions.

Directed DSS

A distributed subsection (DSS) is the action of sending subsections to the database partition that needs to do some work for a parallel query. It also

describes the initiation of subsections with invocation specific values, such as values of variables in an OLTP environment. A *directed DSS* uses the table partition key to direct a query to a single partition. Use this type of query in your application to avoid the coordinator partition overhead required for a query broadcast to all nodes.

An example SELECT statement fragment that can take advantage of directed DSS follows:

```
SELECT ... FROM t1
  WHERE PARTKEY=:hostvar
```

When the coordinator partition receives the query, it determines which partition holds the subset of data for *:hostvar* and directs the query specifically to that partition.

To optimize your application using directed DSS, divide complex queries into multiple simple queries. For example, in the following query the coordinator partition matches the partition key with multiple values. Because the data that satisfies the query lies on multiple partitions, the coordinator partition broadcasts the query to all partitions:

```
SELECT ... FROM t1
  WHERE PARTKEY IN (:hostvar1, :hostvar2)
```

Instead, break the query into multiple SELECT statements (each with a single host variable) or use a single SELECT statement with a UNION to achieve the same result. The coordinator partition can take advantage of simpler SELECT statements to use directed DSS to communicate only to the necessary partitions. The optimized query looks like:

```
SELECT ... AS res1 FROM t1
  WHERE PARTKEY=:hostvar1
  UNION
SELECT ... AS res2 FROM t1
  WHERE PARTKEY=:hostvar2
```

Note that the above technique will only improve performance if the number of selects in the UNION is significantly smaller than the number of partitions.

Using Local Bypass

A specialized form of the directed DSS query accesses data stored only on the coordinator partition. This is called a *local bypass* because the coordinator partition completes the query without having to communicate with another partition.

Local bypass is enabled automatically whenever possible, but you can increase its use by routing transactions to the partition containing the data for that transactions. One technique for doing this is to have a remote client maintain connections to each partition. A transaction can then use the correct

connection based on the input partition key. Another technique is to group transaction by partition and have separate application server for each partition.

In order to determine the number of the partition on which transaction data resides, you can use the `sqlugrpn` API (Get Row Partitioning Number). This API allows an application to efficiently calculate the partition number of a row, given the partitioning key. For more information on the `sqlugrpn` API, refer to the *Administrative API Reference*.

Another alternative is to use the `db2atld` utility to divide input data by partition number and run a copy of the application against each partition. For more information on the `db2atld` utility, refer to the *Command Reference*.

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 53. For more information about preparing an application, see “Creating and Preparing the Source Files” on page 47.

Note: Buffered inserts cause the following steps 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 to a different table
     - PREPARE of the same dynamic statement (by name) doing buffered inserts
     - REDISTRIBUTE NODEGROUP
     - RELEASE SAVEPOINT
     - REORG
     - REVOKE
     - ROLLBACK
     - ROLLBACK TO SAVEPOINT
     - RUNSTATS
     - SAVEPOINT
     - 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 560 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:
   - The target partition receives the row.
   - The target partition inserts the row locally.
   - 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 557.
- 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 the merge. If all the partitions direct their answer sets to the same file on an NFS mount, the results are merged. Note that using NFS without blocking the answer into large buffers results in very poor performance.

```
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);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
   exit(7);
file_write_err:
    printf("Error: file write error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(8);
file_unlock_err:
    printf("Error: file unlock error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
   fcntl(iFileHandle, lock_command, &unlock);
    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 perform 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.

- Create the TEMP table with the NOT LOGGED INITIALLY attribute, then COMMIT the unit of work that created the table to release any acquired catalog locks.
- When you use the TEMP table, you should issue the following statements in a single unit of work:
  1. ALTER TABLE TEMP ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE (to empty the TEMP table and turn logging off)
  2. INSERT INTO TEMP SELECT FIRSTNAME...
  3. COMMIT

  This technique allows you to insert a large answer set into a table without logging and without catalog contention. Note that any error in the unit of work that activated the NOT LOGGED state results in an unusable TEMP table. If this occurs, you will have to drop and recreate the TEMP table. For this reason, you should *not* use this technique to add data to a table that you could not easily recreate.

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.

Creating a Test Environment

You can create a test environment for your partitioned environment applications with DB2 Enterprise Edition, because DB2 Enterprise Edition enforces partitioning key restrictions like DB2 Enterprise - Extended Edition.

1. Create a model of your database design with DB2 Enterprise Edition.
2. Create sample tables with the PARTITIONING KEY clause that you will use to distribute your data across partitions in the production environment.
3. Develop and run your applications against the test database.

DB2 Enterprise Edition enforces the partitioning key constraints consistent with DB2 Enterprise - Extended Edition and provides a useful test environment for your applications.

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 570 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 19. Writing Programs for DB2 Federated Systems

Introduction to DB2 Federated Systems

A *DB2 federated system* is a distributed computing system that consists of:

- A DB2 server, called a *federated server*.
- Multiple semi-autonomous data sources that the federated server sends queries to. Each data source consists of an instance of a relational database management system, plus the database or databases that the instance supports. The data sources in a DB2 federated system can include Oracle instances and instances of the members of the DB2 family.

To client applications, the data sources appear as a single collective database. However, the applications actually interface with a database, called the *federated database*, that is within the federated server. To obtain data from data sources, they submit queries in DB2 SQL to the federated database. DB2 then

distributes the queries to the appropriate data sources, collects the requested data, and returns this data to the applications.

Applications can use DB2 SQL to request values of any data types that DB2 can recognize, except for LOB data types. To write to a data source—for example, to update a data source table—an application must use the data source's own SQL in a special mode called pass-through.

The federated database's system catalog contains information not only about the objects in the database, but also about the data sources and certain tables, views, and functions in them. The catalog, then, contains information about the entire federated system; accordingly, it is called a *global catalog*.

For a high-level overview of DB2 federated systems, see the *Administration Guide: Planning*. For an extended overview, see the *SQL Reference*. For examples of DB2 SQL queries that an application can submit, see "Using Distributed Requests to Query Data Sources" on page 583. For information about Pass-Through, see "Using Pass-Through to Query Data Sources Directly" on page 588.

Accessing Data Source Tables and Views

This section provides information to help you access and use data source tables and views. The topics covered are:

- Nicknames that you assign to the tables and views, so that the federated server can reference them.
- Isolation levels that help you to maintain data integrity at the data source when you access the tables and views

Working with Nicknames

A *nickname* is an identifier by which an application can reference a data source table or view. This section:

- Explains how the global catalog can be supplied with information about tables that you create nicknames for
- Lists considerations and restrictions to remember when you work with nicknames
- Describes parameters that you can set to optimize queries
- Discusses ways to use views referenced by nicknames

Cataloging Information about Data Source Tables and Views

When a nickname is created for a data source table or view, DB2 updates the global catalog with information that the optimizer can use in planning how to retrieve data from the table or view. This information includes, for example, the name of the table or view and the names and attributes of the table's or view's columns.

In the case of a table, the information also includes:

- Statistics (for example, the number of rows and the number of pages on which the rows exist). To ensure that DB2 obtains the latest statistics, it is advisable to run the data source's equivalent of the RUNSTATS command against the table before you create the nickname.
- Descriptions of any indexes that the table has. If the table has no indexes, you can nonetheless supply the catalog with metadata that an index definition typically contains—for example, which column or columns in the table have unique values, and whether any rows are unique. You can generate this metadata, which is collectively called an *index specification*, by running the CREATE INDEX statement against the table's nickname. Be aware that in this case the statement produces only the index specification; it does not create an actual index. For documentation on this statement, see the *SQL Reference*.

To find out what information about a data source table is stored in the global catalog, query the SYSCAT.TABLES and SYSCAT.COLUMNS catalog view. To find out what information about a table's index is stored in the catalog, or what a particular index specification contains, query the SYSCAT.INDEXES catalog view. For descriptions of these views, see the *SQL Reference*. For further discussion about updating the global catalog with information about tables and indexes, see the *Administration Guide: Implementation*.

Considerations and Restrictions

There are several considerations and restrictions to bear in mind when you:

- Define, change, and drop nicknames
- Reference tables and views by their nicknames
- Perform operations on tables and views that are referenced by nicknames

Defining, Changing, and Dropping Nicknames:

- To define a nickname for a table or view, use the CREATE NICKNAME statement. In this statement:
  - You reference an Oracle table or view by its name.
  - You can reference a DB2 family table or view by its name, or, if it has an alias, by this alias.
- You can define more than one nickname for the same table or view. You can also define an alias for a nickname with the CREATE ALIAS statement.
- To change a nickname, you must drop it and then replace it. To drop it, use the DROP NICKNAME statement; to replace it, use the CREATE NICKNAME statement.
- Dropping a nickname causes any views defined using the nickname to be inoperative and invalidates any plans that are dependent upon it.

For documentation on the CREATE NICKNAME, CREATE ALIAS, and DROP NICKNAME statements, see the *SQL Reference*.

Referencing Tables and Views by Nickname:

- After a data source table or view has been given a nickname, you can reference the table or view by that nickname only (except in a pass-through session). For example, if you define the nickname DEPT to represent a table called DB2MVS1.PERSON.DEPT, the statement SELECT * FROM DEPT is allowed, but SELECT * FROM DB2MVS1.PERSON.DEPT is not allowed. In a pass-through session, however, you must access a table or view by its data source name.
- You cannot reference a nickname in the CREATE TRIGGER statement.
- If you reference a nickname in the summary-table-definition clause of the CREATE TABLE statement, you must also specify the DEFINITION ONLY keywords in this clause.

Performing Operations on Tables and Views That Have Nicknames:

- The COMMENT ON statement is valid against a nickname and columns that are defined on a nickname. This statement updates the global catalog; it does not update data source catalogs.
- GRANT and REVOKE statements are valid against a nickname for certain privileges and for all users and groups. However, DB2 does not issue a corresponding GRANT or REVOKE against the table or view that the nickname references. For more information about nickname privileges, see the *Administration Guide: Planning*.
- Data sources are read-only. Therefore:
  - INSERT, UPDATE, and DELETE statements are not valid against nicknames.
  - A view that contains a UNION ALL clause for a nickname cannot be updated.
- You cannot run the DB2 utilities (RUNSTATS, IMPORT, EXPORT and so on) against nicknames.

Defining Column Options

When you define a nickname for a table or view, you can provide the global catalog with information about particular columns in the table or view. You specify this information in the form of values that you assign to parameters called *column options.* You can specify any of these values in either upper- or lowercase. Table 27 on page 577 describes the column options and their values.

Table 27. Column Options and Their Settings

Option	Valid Settings		Default Setting
numeric_string	'y'	Yes, this column contains only strings of numeric data. IMPORTANT: If this column contains only numeric strings followed by trailing blanks, it is inadvisable to specify 'y'.	'n'
	'n'	No, this column is not limited to strings of numeric data.	
	By setting numeric_string to 'y' for a column, you are informing the optimizer that this column contains no blanks that could interfere with sorting of the column's data.		
varchar_no_trailing_blanks	Indicates whether trailing blanks are absent from a specific VARCHAR column:		'n'
	'y'	Yes, trailing blanks are absent from this VARCHAR column.	
	'n'	No, trailing blanks are not absent from this VARCHAR column.	
	If data source VARCHAR columns contain no padded blanks, then the optimizer's strategy for accessing them depends in part on whether they contain trailing blanks. By default, the optimizer "assumes" that they actually do contain trailing blanks. On this assumption, it develops an access strategy that involves modifying queries so that the values returned from these columns are the ones that the user expects. If, however, a VARCHAR column has no trailing blanks, and you let the optimizer know this, it can develop a more efficient access strategy. To tell the optimizer that a specific column has no trailing blanks, specify that column in the ALTER NICKNAME statement (for guidelines, see the *SQL Reference*).		

You set column options in the ALTER NICKNAME statement. For information about this statement, see the *SQL Reference*.

Using Nicknames with Views

You can use nicknames with views in two main ways:

- You can create nicknames for data source views. The federated server treats the nickname of a data source view the same way it treats the nickname of a data source table.
- You can create federated database views of data source tables and views that have nicknames. For example, because the federated server can accommodate a join of base tables at different locations, you can easily

define federated database views of base tables that reside at different data sources. Such multi-location views offer a high degree of data independence for a globally integrated database, just as views defined on multiple local tables do for centralized relational database managers. This global view mechanism is one way in which the federated server offers a high degree of data independence.

The action of creating a federated database view of data source data is sometimes called "creating a view on a nickname". This phrase reflects the fact that for the view to be created, the CREATE VIEW statement's fullselect must reference the nickname of each table and view that the view is to contain.

Views do not have statistics or indexes of their own because they are not actual tables located in a database. This statement is true even when a view is identical in structure and content to a single base table. For more information about statistics and indexes, see *Administration Guide: Implementation*.

Using Isolation Levels to Maintain Data Integrity

You can maintain data integrity for a data source table by requesting that the table's rows be locked at a specific isolation level. For example, to ensure that you have sole access to a row, you would specify the repeatable read (RR) isolation level for that row.

The federated server maps the isolation level you request to a corresponding one at the data source. To illustrate this, Table 28 lists:

- The isolation levels that you can request. They are:

  CS Cursor stability

  RR Repeatable read

  RS Read stability

  UR Uncommitted read

- The Oracle isolation levels that the requested levels map to.

Table 28. Comparable Isolation Levels between the Federated Server and Oracle Data Sources.

Federated Server (DB2)	CS	RR	RS	UR
Oracle	Default	Transaction read-only	Transaction read-only	Same as cursor stability

Working with Data Type Mappings

When you create a nickname for a data source table, DB2 populates the global catalog with information about the table. This information includes, but is not limited to, the nickname, the table's name, all column names and, for each column:

- The data type that was defined for the column at the data source. (This section calls this type a *remote* type.)
- A corresponding data type that is supported by DB2 and registered to the federated database. (This section calls this type a *local* type.)

This section explains how DB2 uses data type mappings to determine what DB2-supported data type should be defined for the column of a data source table. The section then discusses data type mappings (sometimes called "type mappings", for short) in two subsections. The first describes default mappings; the second shows how you can override default mappings and create new mappings.

How DB2 Determines What Data Types to Define Locally

How does DB2 determine what local type to use for a remote column? It consults a mapping between the column's type at the data source and a comparable local type, and chooses the latter. For example, in a default mapping supplied by DB2, the DB2 for VSE & VM data type CHAR, which supports up to 254 bytes, points to the DB2 data type CHAR. So if you create a nickname for a DB2 for VSE & VM table, and column C1 of the table has a data type of CHAR with a maximum length of 200, then, unless you override the default, the DB2 type CHAR is defined locally for C1.

Default Data Type Mappings

Because of differences between RDBMSs, a default mapping between a data source data type and a federated server data type is not always one-to-one. However, the mapping is close enough to ensure that all requested values are returned.

For example, there is a default type mapping between:

- The Oracle type NUMBER(9,0) (where 9 is the maximum precision and 0 the maximum scale)
- The DB2 type INTEGER, with a maximum length of 4 bytes

Suppose that you create a nickname for an Oracle table that has a column C2 with a type of NUMBER(9,0). If you do not change the default mapping, the type for C2 will be locally defined as INTEGER. And because the 4 bytes of INTEGER support a maximum precision of 10, you can be sure that all values of C2 will be returned when C2 is queried from the federated server.

For listings of the default data type mappings, see the *SQL Reference*.

How You Can Override Default Type Mappings and Create New Ones

As the preceding example indicates, the local type and remote type in a default mapping are similar enough to ensure that when you query remote columns for which the remote type is defined, all values that conform to both types will be returned. But sometimes, you might require an alternative mapping. Consider these scenarios:

Defining a Type Mapping That Applies to One or More Data Sources

Certain columns in three tables in an Oracle data source have a data type DATE for time stamps. In a default mapping, this type points to the local DB2 type TIMESTAMP. So if you were to create nicknames for the three tables without changing the default, TIMESTAMP would be defined locally for these columns, and DB2 queries of the columns would yield time stamps. But suppose that you want such queries to yield times only. You could then map Oracle DATE to the DB2 type TIME, overriding the default. That way, when you create the nicknames, TIME, not TIMESTAMP, would be defined locally for the columns. As a result, when you query them, only the time portion of the time stamps would be returned. To override the default type mapping, you would use the CREATE TYPE MAPPING statement.

In the CREATE TYPE MAPPING statement, you can indicate whether the new mapping that you want is to apply to a specific data source (for example, a data source that a department in your organization uses) or to all data sources of a specific type (for example, all Oracle data sources), or to all data sources of a specific version of a type (for example, all Oracle 8.0.3 data sources).

Changing a Type Mapping for a Specific Table

You can change the local type in a type mapping for a specific table. For example, Oracle data type NUMBER(32,3) maps by default to the DB2 data type DOUBLE, a floating decimal data type. Suppose that in an Oracle table for employee information, a column BONUS was defined with a data type of NUMBER(32,3). Because of the mapping, a query of BONUS could return values that look like this:

```
5.0000000000000E+002
1.0000000000000E+003
```

where +002 signifies that the decimal point should be moved two places to the right, and +003 signifies that the decimal point should be moved three places to the right.

So that queries of BONUS can return values that look like dollar amounts, you could, for this particular table, remap NUMBER(32,3) to a DB2 DECIMAL type with a precision and scale that reflect the format of actual bonuses. For example, if you knew that the dollar portion of the bonuses would not exceed

six figures, you could remap NUMBER(32,3) to DECIMAL(8,2). Under the constraint of this new mapping, a query of BONUS would return values like this:

```
 500.00
1000.00
```

To change the type mapping for a column of a specific table, use the ALTER NICKNAME statement. With this statement, you can change the type defined locally for a column of a table for which a nickname has been defined.

Large Object (LOB) Support

In a federated database system, you can access and manipulate LOBs at remote data sources. Because LOBs can be very large, transferring LOBs from a remote data source can be time consuming. The DB2 federated database attempts to minimize transferring LOB data from the data sources, and also attempts to deliver requested LOB data directly from the data source to the requesting application without materializing the LOB at DB2.

This section discusses:

- How DB2 retrieves LOBs
- How applications can use LOB locators
- Restrictions on LOBs
- Mappings between LOB and non-LOB data types
- Tuning the system

How DB2 Retrieves LOBs

DB2 federated systems use two mechanisms to retrieve LOBs: LOB streaming and LOB materialization.

LOB Streaming

In LOB streaming, LOB data is retrieved in stages. DB2 uses LOB streaming for data in result sets of queries that are completely pushed down. For example, consider the following query:

```
SELECT empname, picture FROM orc_emp_table WHERE empno = '01192345'
```

where *picture* represents a LOB column and *orc_emp_table* represents a nickname referencing an Oracle table containing employee data. The DB2 query processor marks the *picture* column for streaming if it decides to run the entire query at the Oracle data source. At execution time, if DB2 notes that a LOB is marked for streaming, it retrieves the LOB in stages from the data source. DB2 then transfers the data to the application memory space.

LOB Materialization

In LOB materialization, the remote LOB data is retrieved by DB2 and stored locally at the federated server. DB2 uses LOB materialization when:

- The LOB column cannot be deferred or streamed.
- A function must be applied to a LOB column locally, before the data is transferred. This happens when DB2 compensates for functions not available at a remote data source. For example, Microsoft SQL Server does not provide a SUBSTR function for LOB columns. To compensate, DB2 materializes the LOB column locally and applies the DB2 SUBSTR function to the retrieved LOB.

How Applications can use LOB locators

Applications can request LOB locators for LOBs stored in remote data sources. 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). For additional information about LOB locators, see "Understanding Large Object Locators" on page 351.

DB2 can retrieve LOBs from remote data sources, store them at DB2, and then issue a LOB locator against the stored LOB. LOB locators are released when:

- Applications issue ″FREE LOCATOR″ SQL statements.
- Applications issue COMMIT statements.
- DB2 is restarted.

Restrictions on LOBs

When using and retrieving LOBs, consider that:

- DB2 is unable to bind remote LOBs to a file reference variable.
- LOBs are not supported in pass-through mode.

Mappings Between LOB and Non-LOB Data Types

There are a few cases in which you can map a DB2 LOB data type to a non-LOB data type at a data source. When you need to create a mapping between a column with a DB2 LOB type and its counterpart column at a data source, it is recommended that you use a LOB data type as a counterpart if at all possible.

To create a mapping, use the create type mapping DDL statement. For example:

```
CREATE TYPE MAPPING my_oracle_lob FROM sysibm.clob TO SERVER TYPE oracle TYPElong
```

where:

my_oracle_lob
Is the name of the type mapping.

sysibm.clob
Is the DB2 CLOB data type.

oracle Is the type of server you are connecting to.

long Is the Oracle data type counterpart.

Using Distributed Requests to Query Data Sources

Queries submitted to the federated database can request results that are yielded by a single data source; but typically they request results that are yielded by multiple data sources. Because a typical query is distributed to multiple data sources, it is called a *distributed request.*

This section:

- Illustrates ways to code distributed requests
- Introduces you to a way to abet optimization of certain distributed requests

Coding Distributed Requests

In general, a distributed request uses one or more of three SQL conventions to specify where data is to be retrieved from: subqueries, set operators, and join subselects. This section provides examples within the context of the following scenario: A federated server is configured to access a DB2 Universal Database for OS/390 data source, a DB2 Universal Database for AS/400 data source, and an Oracle data source. Stored in each data source is a table that contains employee information. The federated server references these tables by nicknames that point to where the tables reside: UDB390_EMPLOYEES, AS400_EMPLOYEES, and ORA_EMPLOYEES. (Nicknames do not have to reference data sources; the ones in this scenario do so only to underline the point that the tables reside in different RDBMSs.) In addition to ORA_EMPLOYEES, the Oracle data source has a table, nicknamed ORA_COUNTRIES, that contains information about the countries that the employees live in.

A Request with a Subquery

Table AS400_EMPLOYEES contains the phone numbers of employees who live in Asia. It also contains the country codes associated with these phone numbers, but it does not list the countries that the codes represent. Table ORA_COUNTRIES, however, does list both codes and countries. The following query uses a subquery to find out the country code for China; and it uses SELECT and WHERE clauses to list those employees in AS400_EMPLOYEES whose phone numbers require this particular code.

```
SELECT name, telephone
   FROM djadmin.as400_employees
   WHERE country_code IN
```

```
(SELECT country_code
   FROM djadmin.ora_countries
   WHERE country_name = 'CHINA')
```

A Request with Set Operators

The federated server supports three set operators:

- UNION

  Use this set operator to combine the rows that satisfy any of two or more SELECT statements.

- EXCEPT

  Use this set operator to retrieve those rows that satisfy the first SELECT statement but not the second.

- INTERSECT

  Use this set operator to retrieve those rows that satisfy both SELECT statements.

All three set operators might have the ALL operand to indicate that duplicate rows are not be removed from the result, thus eliminating the need for an extra sort.

The following query retrieves all employee names and country codes that are present in both the AS400_EMPLOYEES and UDB390_EMPLOYEES tables, even though each table resides in a different data source.

```
SELECT name, country_code
   FROM as400_employees
INTERSECT
SELECT name, country_code
   FROM udb390_employees
```

A Request for a Join

A relational join produces a result set that contains a combination of columns retrieved from two or more tables. Be aware that you should specify conditions to limit the size of the result set's rows.

The query below combines employee names and their corresponding country names by comparing the country codes listed in two tables. Each table resides in a different data source.

```
SELECT t1.name, t2.country_name
   FROM djadmin.as400_employees t1, djadmin.ora_countries t2
   WHERE t1.country_code = t2.country_code
```

Using Server Options to Facilitate Optimization

Federated system users can use parameters called *server options* to supply the global catalog with information that applies to a data source as a whole, or to control interaction between DB2 and a data source. For example, to catalog

the identifier of the instance that serves as the basis of a data source, the database administrator assigns that identifier as a value to the server option "node".

Several server options address a major area of interaction between DB2 and data sources: optimization of queries. For example, just as you can use the column option "varchar_no_trailing_blanks" to inform the DB2 optimizer of specific data source VARCHAR columns that have no trailing blanks, so can you use a server option—also called "varchar_no_trailing_blanks"—to inform the optimizer of data sources whose VARCHAR columns are all free of trailing blanks. For a summary of how such information helps the optimizer to create an access strategy, see Table 27 on page 577.

In addition, you can set the server option "plan_hints" to a value that enables DB2 to provide Oracle data sources with statement fragments, called *plan hints*, that help Oracle optimizers do their job. Specifically, plan hints can help an optimizer to decide matters such as which index to use in accessing a table, and which table join sequence to use in retrieving data for a result set.

Typically, the database administrator sets server options for a federated system. However, a programmer can make good use of those options that help to optimize queries. For example, suppose that for data sources ORACLE1 and ORACLE2, the plan_hints server option is set to its default, 'n' (no, do not furnish this data source with plan hints). Also suppose that you write a distributed request for data from ORACLE1 and ORACLE2, and that you expect that plan hints would help the optimizers at these data sources improve their strategies for accessing this data. You could override the default with a setting of 'y' (yes, furnish the plan hints) while your application is connected to the federated database. When the connection is completed, the setting would automatically revert to 'n'.

To enforce a server option setting for the duration of a connection to the federated database, use the SET SERVER OPTION statement. To ensure that the setting takes effect, you must specify the statement right after the CONNECT statement. In addition, it is advisable to prepare the statement dynamically.

For documentation of the SET SERVER OPTION statement, see the *SQL Reference*. For descriptions of all server options and their settings, see the *Administration Guide: Implementation*.

Invoking Data Source Functions

This section explains how you can:

- Enable DB2 to invoke a data source function that it does not recognize
- Help to reduce the overhead consumed when DB2 invokes a function
- Specify names of functions that you want to map to one another
- Discontinue the use of a mapping between functions

Enabling DB2 to Invoke Data Source Functions

At times you might want DB2 to invoke a data source function that it does not recognize. Such a function might be a user-defined function or a new built-in function that is unknown to DB2.

Before DB2 can access a data source function that it does not recognize, you must create a mapping between this function and a counterpart that is stored in the federated database. To create the mapping, select the counterpart and submit the DDL statement for creating the mapping. This statement is called CREATE FUNCTION MAPPING.

The counterpart can be an existing function or function template, or a function or function template that you create. (A *function template* is a partial function that has no executable code.) You can create a function or function template with the CREATE FUNCTION statement.

The data source function and its federated database counterpart should correspond in the following ways:

- Both should have the same number of input parameters.
- The data types of the input parameters of the data source function should be compatible with the data types of the input parameters of the federated database counterpart.

For documentation on the CREATE FUNCTION MAPPING and CREATE FUNCTION statements, see the *SQL Reference*.

Reducing the Overhead of Invoking a Function

The DDL for mapping a federated server function to a data source function—the CREATE FUNCTION MAPPING statement—can include estimated statistics on the overhead that would be consumed when the data source function is invoked. For example, the statement can specify the estimated number of instructions that would be required to invoke the data source function, and the estimated number of I/Os that would be expended for each byte of the argument set that is passed to this function. These estimates are stored in the global catalog; you can see them in the SYSCAT.FUNCMAPOPTIONS view. In addition, if a DB2 function (rather than a function template) participates in the mapping, the catalog contains

estimates of overhead that would be consumed when this function is invoked. You can see these latter estimates in the SYSCAT.FUNCTIONS view.

After the mapping is created, you can submit distributed requests that reference the DB2 function. For example, if you mapped a DB2 user-defined function called DOLLAR to an Oracle user-defined function called US_DOLLAR, your request would specify DOLLAR rather than US_DOLLAR. When the request is processed, the optimizer evaluates multiple access strategies. Some of them reflect the estimated overhead of invoking the DB2 function; others reflect the estimated overhead of invoking the data source function. The strategy that is expected to cost the least amount of overhead is the one that is used.

If any estimates of consumed overhead change, you can record the change in the global catalog. To record new estimates for the data source function, first drop or disable the function mapping (for information about how to do this, see "Discontinuing Function Mappings" on page 588). Then recreate the mapping with the CREATE FUNCTION MAPPING statement, specifying the new estimates in the statement. When you run the statement, the new estimates will be added to the SYSCAT.FUNCTIONS catalog view. To record changed estimates for the DB2 function, update the SYSSTAT.FUNCTIONS catalog view directly.

You specify estimated statistics in the CREATE FUNCTION MAPPING statement by assigning them as values to parameters called *function mapping options*. Table 29 describes these options and their values.

Table 29. Function Mapping Options and Their Settings

Option	Valid Settings	Default Setting
ios_per_invoc	Estimated number of I/Os per invocation of a data source function.	'0'
insts_per_invoc	Estimated number of instructions processed per invocation of the data source function.	'450'
ios_per_argbyte	Estimated number of I/Os expended for each byte of the argument set that is passed to the data source function.	'0'
insts_per_argbyte	Estimated number of instructions processed for each byte of the argument set that is passed to the data source function.	'0'
percent_argbytes	Estimated average percent of input argument bytes that the data source function will actually read.	'100'
initial_ios	Estimated number of I/Os performed the first and last time that the data source function is invoked.	'0'
initial_insts	Estimated number of instructions processed the first and last time that the data source function is invoked.	'0'

For more information about the DROP FUNCTION MAPPING statement, the SYSCAT.FUNCTIONS and SYSSTAT.FUNCTIONS views, and the SYSCAT.FUNCMAPOPTIONS view, see the *SQL Reference*.

Specifying Function Names in the CREATE FUNCTION MAPPING Statement

How you code the CREATE FUNCTION MAPPING statement depends on part on whether the names of the objects that you are mapping together are the same or different. If you are creating a mapping between two functions (or a function template and a function) that have the same name, you must assign this name to the *function-name* parameter.

But if the names differ, then:

- Assign the name of the federated database function or function template to the *function-name* parameter.
- Specify a function mapping option called "remote_name" and assign the name of the data source function to this option. The name must have fewer than 255 characters.

Discontinuing Function Mappings

If you want to discontinue using a function mapping, follow these guidelines:

- If the mapping is listed in the SYSCAT.FUNCMAPPINGS catalog view, delete the mapping. You do this with the DROP FUNCTION MAPPING statement.
- To discontinue a default mapping that is not listed in the SYSCAT.FUNCMAPPINGS view, you disable the mapping. You do this in the CREATE FUNCTION MAPPING statement by setting a function mapping option called "disable" to 'y' (yes, disable this function mapping). The default is 'n'.

Using Pass-Through to Query Data Sources Directly

You can use a facility called *pass-through* to query a data source in the SQL that is native to that data source. This section:

- States what kind of SQL statements a federated server and its associated data sources process in pass-through sessions.
- Lists considerations and restrictions to be aware of when you use pass-through.

SQL Processing in Pass-Through Sessions

The following rules specify whether an SQL statement is processed by DB2 or by a data source:

- If a static statement is submitted in a pass-through session, it is sent to the federated server for processing.
- If, in a pass-through session, you want to submit an SQL statement to a data source for processing, you must prepare it dynamically in the session and have it executed while the session is still open.
  - If you are submitting a SELECT statement, prepare it with the PREPARE statement, and then use the OPEN, FETCH, and CLOSE statements to access the results of your query.
  - For a supported statement other than SELECT, you have two options:
    - Use the PREPARE statement to prepare the supported statement, and use the EXECUTE statement to execute it.
    - Use the EXECUTE IMMEDIATE statement to prepare and execute the statement.
- If you issue the COMMIT or ROLLBACK command during a pass-through session, this command will complete the current unit of work (UOW).

Considerations and Restrictions

There are a number of considerations and restrictions to bear in mind when you use pass-through. Some of them are of a general nature; others apply to Oracle data sources only.

Using Pass-Through with All Data Sources

The following information applies to all data sources:

- Statements prepared within a pass-through session must be executed within the same pass-through session. Statements prepared within a pass-through session, but executed outside of the same pass-through session, will fail (SQLSTATE 56098).
- You can use pass-through to write to data sources; for example, to insert, update, and delete table rows. But note that you cannot use WHERE CURRENT OF conditions in UPDATE and DELETE statements within a pass-through session.
- An application can have several SET PASSTHRU statements in effect at the same time to different data sources. Although the application might have issued multiple SET PASSTHRU statements, the pass-through sessions are not truly nested. The federated server will not pass through one data source to access another. Rather, the server accesses each data source directly.
- If multiple pass-through sessions are open at the same time, be sure to issue a COMMIT each time you want to conclude a unit of work in each session. Then, when you need to terminate the sessions, you can do so with a single SET PASSTHRU RESET statement.
- Host variables defined in SQL statements within a pass-through session must take the form :H*n* where H is uppercase and *n* is a unique whole number. The values of *n* must be numbered consecutively beginning with zero.

- You cannot pass through to more than one data source at a time.
- Pass-through does not support stored procedure calls.
- Pass-through does not support the SELECT INTO statement.

Using Pass-Through with Oracle Data Sources

The following information applies to Oracle data sources:

- The following restriction applies when a remote client issues a SELECT statement from a command line processor (CLP) in pass-through mode: If the client code is a DB2 Application Development Client prior to DB2 Universal Database Version 5, the SELECT will elicit an SQLCODE -30090 with reason code 11. To avoid this error, remote clients must use a DB2 Application Development Client that is at Version 5 or greater.
- Any DDL statement issued against an Oracle server is performed at parse time and is not subject to transaction semantics. The operation, when complete, is automatically committed by Oracle. If a rollback occurs, the DDL is not rolled back.
- When you issue a SELECT statement from raw data types, use the RAWTOHEX function to receive the hexadecimal values. When you perform an INSERT into raw data types, provide the hexadecimal representation.

Part 6. Language Considerations

Chapter 20. Programming in C and C++

Programming Considerations for C and C++

Special host language programming considerations are discussed in the following sections. 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 for C and C++

The following sections describe the C/C++ language restrictions.

Trigraph Sequences for C and C++

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.

Input and Output Files for C and C++

By default, the input file can have the following extensions:

.sqc For C files on all supported platforms

.sqC For C++ files on UNIX platforms

.sqx For C++ files on OS/2 and Windows 32-bit operating systems

By default, the corresponding precompiler output files have the following extensions:

.c For C files on all supported platforms

.C For C++ files on UNIX platforms

.cxx For C++ files on OS/2 and Windows 32-bit operating systems

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.

Include Files for C and C++

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 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 419.

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.

Including Files in C and C++

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 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 quotation marks, as above, the C precompiler searches for `payroll.sqc`, then `payroll.h`, in each directory in which it looks. On UNIX operating systems, the 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-32 bit operating systems, the 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 quotation marks, as above, no extension is added to the name.

  If the file name in quotation marks 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.

However, if you specify the PREPROCESSOR option, all the #line macros generated by the precompiler reference the preprocessed file from the external C preprocessor. For more information about the PREPROCESSOR option, see "C Macro Expansion" on page 613.

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.

Embedding SQL Statements in C and C++

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 before the statement initializer or after the statement terminator.
- 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 quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, 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 in C and C++

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. Follow the rules described in the following sections when naming, declaring, and using host variables.

Naming Host Variables in C and C++

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, sql, DB2, and db2, 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 in C and C++

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. Typedefs for host variable types are not allowed. Host variables can be grouped into a single host structure. For more

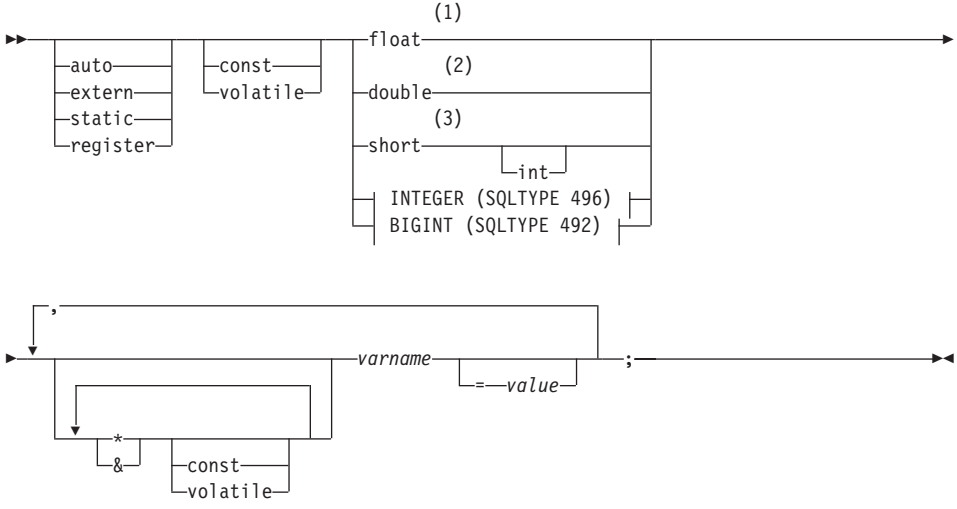
information on host structures, see “Host Structure Support in C and C++” on page 614. You can declare C++ class data members as host variables. For more information on classes, see “Using Class Data Members as Host Variables in C and C++” on page 620.

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.

For information on declaring host variables for structured types, see “Declaring Structured Type Host Variables” on page 348.

“Syntax for Numeric Host Variables in C or C++” shows the syntax for declaring numeric host variables in C or C++.

Syntax for Numeric Host Variables in C or C++



INTEGER (SQLTYPE 496)

BIGINT (SQLTYPE 492)

Notes:

1 REAL (SQLTYPE 480), length 4

2 DOUBLE (SQLTYPE 480), length 8

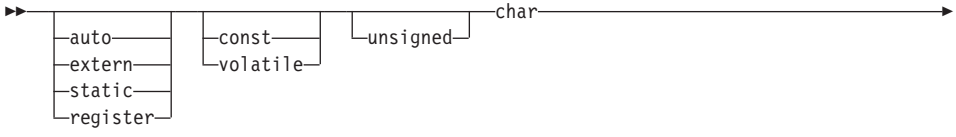
3 SMALLINT (SQLTYPE 500)

4 For maximum application portability, use sqlint32 and sqlint64 for INTEGER and BIGINT host variables, respectively. By default, the use of long host variables results in precompile error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

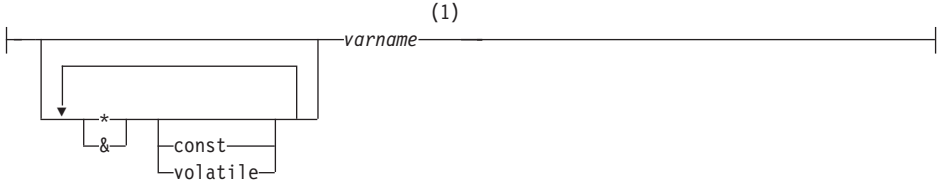
5 For maximum application portability, use sqlint32 and sqlint64 for INTEGER and BIGINT host variables, respectively. To use the BIGINT data type, your platform must support 64 bit integer values. By default, the use of long host variables results in precompile error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

"Form 1: Syntax for Fixed and Null-terminated Character Host Variables in C/C++" shows the syntax for declaring fixed and null-terminated character host variables in C or C++.

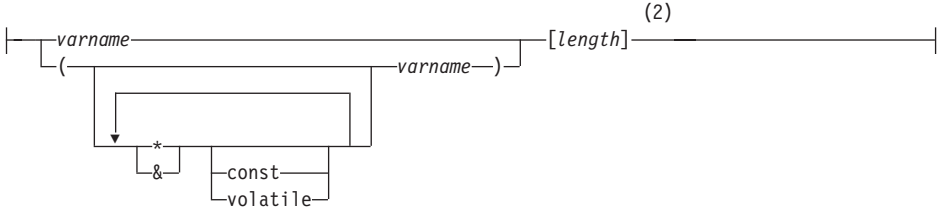
Form 1: Syntax for Fixed and Null-terminated Character Host Variables in C/C++



CHAR



C String



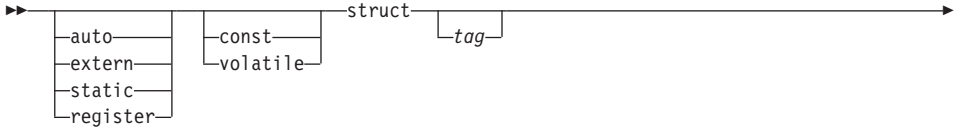
Notes:

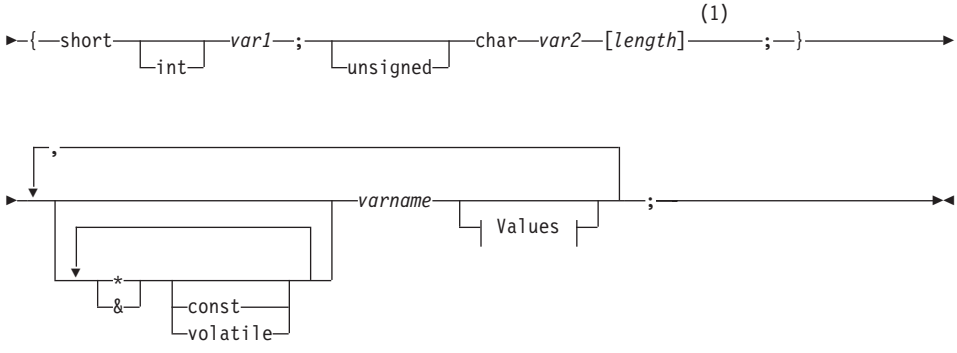
1 CHAR (SQLTYPE 452), length 1

2 Null-terminated C string (SQLTYPE 460); length can be any valid constant expression

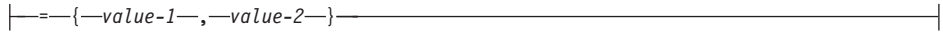
"Form 2: Syntax for Variable Length Character Host Variables in C/C++" shows the syntax for declaring variable length character host variables in C or C++.

Form 2: Syntax for Variable Length Character Host Variables in C/C++





Values



Notes:

1 In form 2, length can be any valid constant expression. Its value after evaluation determines if the host variable is VARCHAR (SQLTYPE 448) or LONG VARCHAR (SQLTYPE 456).

Variable Length Character Host Variable Considerations:

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 value for the length specifier after evaluation must be no greater than 32672, and the string contained by the variable should be null-terminated.
3. If **form 2** is used, the value for the length specifier after evaluation 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 in C and C++" on page 619 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 in C and C++

Indicator variables should be declared as a `short` data type.

Graphic Host Variable Declarations in C or C++

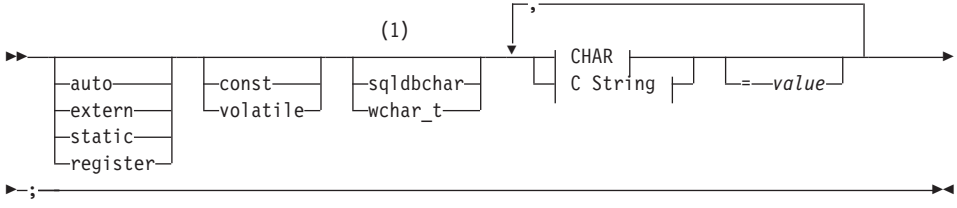
Graphic host variable declarations can take one of three forms:

- Single-graphic form
- null-terminated graphic form
- VARGRAPHIC structured form

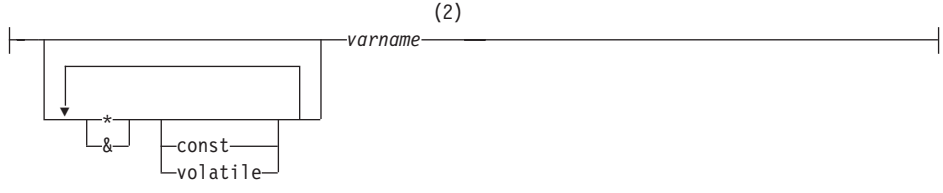
For details on using graphic host variables, see "Handling Graphic Host Variables in C and C++" on page 621.

"Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)" shows the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.

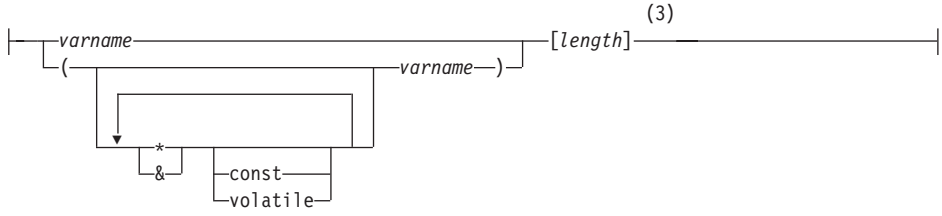
Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)



CHAR



C String



Notes:

1 To determine which of the two graphic types should be used, see "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622.

2 GRAPHIC (SQLTYPE 468), length 1

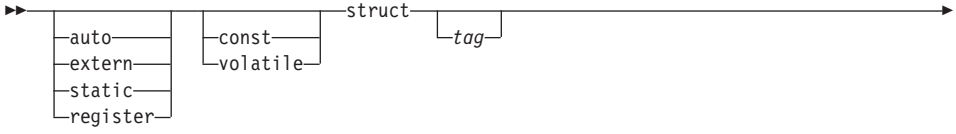
3 Null-terminated graphic string (SQLTYPE 400)

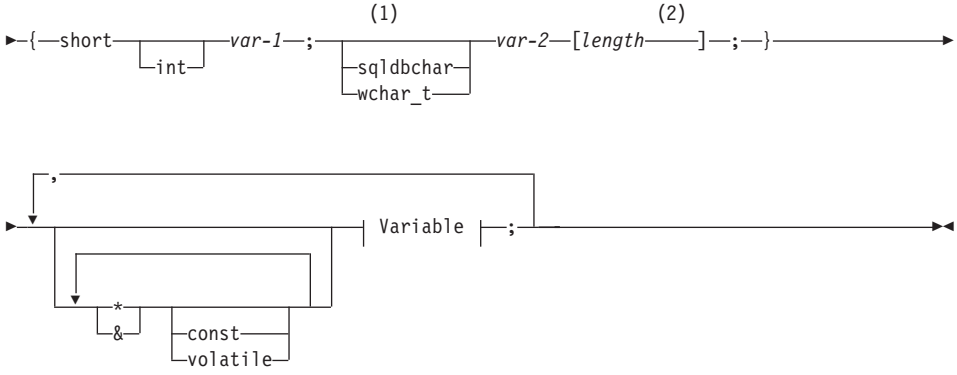
Graphic Host Variable Considerations:

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* can be any valid constant expression, and its value after evaluation must be greater than or equal to 1 and not greater than the maximum length of VARGRAPHIC, which is 16 336.
4. Null-terminated graphic strings are handled differently depending on the value of the standards level precompile option setting. See "Null-terminated Strings in C and C++" on page 617 for details.

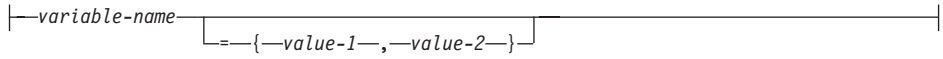
"Syntax for Graphic Declaration (VARGRAPHIC Structured Form)" shows the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

Syntax for Graphic Declaration (VARGRAPHIC Structured Form)





Variable:



Notes:

1 To determine which of the two graphic types should be used, see "Selecting the wchar_t or sqldbchar Data Type in C and C++" on page 622.

2 length can be any valid constant expression. Its value after evaluation determines if the host variable is VARGRAPHIC (SQLTYPE 464) or LONG VARGRAPHIC (SQLTYPE 472). The value of length must be greater than or equal to 1 and not greater than the maximum length of LONG VARGRAPHIC which is 16350.

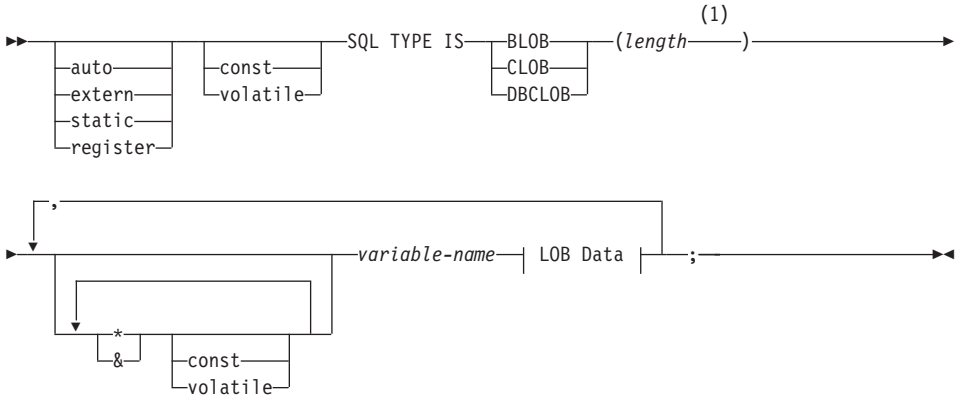
Graphic Declaration (VARGRAPHIC Structured Form) Considerations:

1. *var-1* and *var-2* must be simple variable references (no operators) and cannot be used as host variables.
2. *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.
3. The struct *tag* can be used to define other data areas, but itself cannot be used as a host variable.

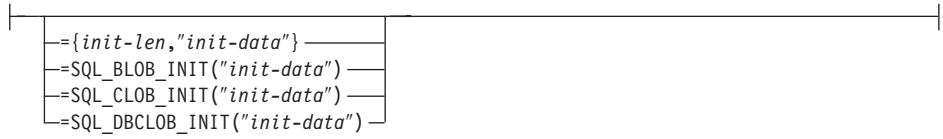
LOB Data Declarations in C or C++

"Syntax for Large Object (LOB) Host Variables in C/C++" on page 609 shows the syntax for declaring large object (LOB) host variables in C or C++.

Syntax for Large Object (LOB) Host Variables in C/C++



LOB Data



Notes:

1 length can be any valid constant expression, in which the constant K, M, or G can be used. The value of length after evaluation for BLOB and CLOB must be 1 <= length <= 2 147 483 647. The value of length after evaluation for DBCLOB must 1 <= length <= 1 073 741 823.

LOB Host Variable Considerations:

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. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G may be in mixed case.
3. The maximum length allowed for the initialization string, *"init-data"*, is 32702 bytes including string delimiters (the same as the existing limit on C/C++ strings within the precompiler).
4. The initialization length, *init-len*, must be a numeric constant (i.e. it cannot include K, M, or G).
5. A length for the LOB must be specified; that is, the following declaration is not permitted:

```
SQL TYPE IS BLOB my_blob;
```

6. If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code.
7. 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.
8. 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 {
      sqluint32        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 {
     sqluint32        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 {
     sqluint32        length;
     sqldbchar        data[30000];
} my_dbclob1;
```

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

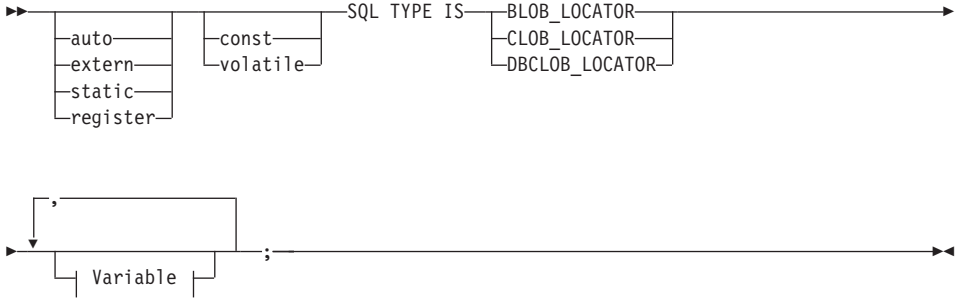
Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

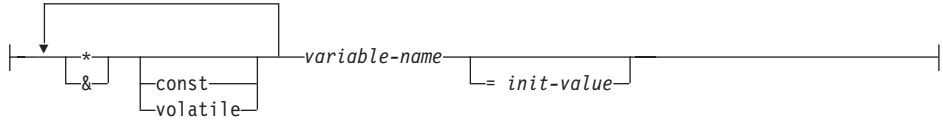
LOB Locator Declarations in C or C++

"Syntax for Large Object (LOB) Locator Host Variables in C/C++" shows the syntax for declaring large object (LOB) locator host variables in C or C++.

Syntax for Large Object (LOB) Locator Host Variables in C/C++



Variable



LOB Locator Host Variable Considerations:

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;
```

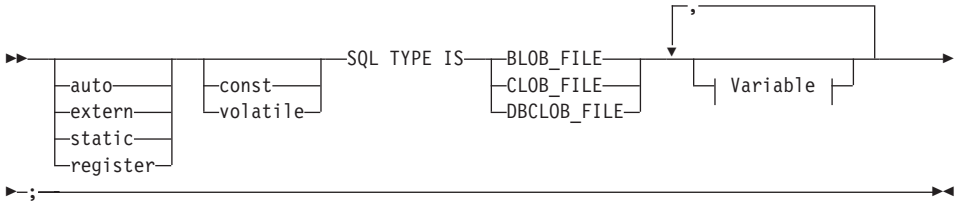
Results in the generation of the following declaration:

```
sqlint32 my_locator;
```

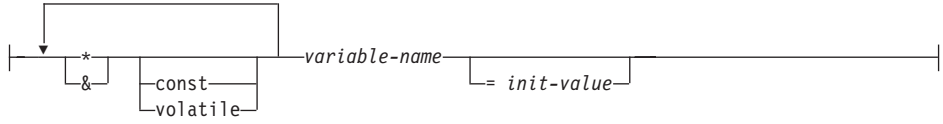
File Reference Declarations in C or C++

"Syntax for File Reference Host Variables in C/C++" shows the syntax for declaring file reference host variables in C or C++.

Syntax for File Reference Host Variables in C/C++



Variable



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 {
     sqluint32        name_length;
     sqluint32        data_length;
     sqluint32        file_options;
              char    name[255];
} my_file;
```

Initializing Host Variables in C and C++

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;
```

C Macro Expansion

The C/C++ precompiler cannot directly process any C macro used in a declaration within a declare section. Instead, you must first preprocess the source file with an external C preprocessor. To do this, specify the exact command for invoking a C preprocessor to the precompiler through the PREPROCESSOR option.

When you specify the PREPROCESSOR option, the precompiler first processes all the SQL INCLUDE statements by incorporating the contents of all the files referred to in the SQL INCLUDE statement into the source file. The precompiler then invokes the external C preprocessor using the command you specify with the modified source file as input. The preprocessed file, which the precompiler always expects to have an extension of ″.i″, is used as the new source file for the rest of the precompiling process.

Any #line macro generated by the precompiler no longer references the original source file, but instead references the preprocessed file. In order to relate any compiler errors back to the original source file, retain comments in the preprocessed file. This helps you to locate various sections of the original source files, including the header files. The option to retain comments is commonly available in C preprocessors, and you can include the option in the command you specify through the PREPROCESSOR option. You should not have the C preprocessor output any #line macros itself, as they may be incorrectly mixed with ones generated by the precompiler.

Notes on Using Macro Expansion:

1. The command you specify through the PREPROCESSOR option should include all the desired options but not the name of the input file. For example, for IBM C on AIX you can use the option:

   ```
   xlC -P -DMYMACRO=1
   ```

2. The precompiler expects the command to generate a preprocessed file with a `.i` extension. However, you cannot use redirection to generate the preprocessed file. For example, you **cannot** use the following option to generate a preprocessed file:

   ```
   xlC -E > x.i
   ```

3. Any errors the external C preprocessor encounters are reported in a file with a name corresponding to the original source file but with a `.err` extension.

For example, you can use macro expansion in your source code as follows:

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
  struct
  {
    short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

The previous declarations resolve to the following after you use the PREPROCESSOR option:

```
EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
  struct
  {
    short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

Host Structure Support in C and C++

With host structure support, the C/C++ precompiler allows host variables to be grouped into a single host structure. This provides a shorthand for referencing that same set of host variables in an SQL statement. For example, the following host structure can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
```

```
    short   years;
    double salary;
  } info;
} staff_record;
```

The fields of a host structure can be any of the valid host variable types. These include all numeric, character, and large object types. Nested host structures are also supported up to 25 levels. In the example above, the field `info` is a sub-structure, whereas the field `name` is not, as it represents a VARCHAR field. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. Pointer to host structure is also supported.

There are two ways to reference the host variables grouped in a host structure in an SQL statement:

1. The host structure name can be referenced in an SQL statement.

   ```
   EXEC SQL SELECT id, name, years, salary
        INTO :staff_record
        FROM staff
        WHERE id = 10;
   ```

   The precompiler converts the reference to `staff_record` into a list, separated by commas, of all the fields declared within the host structure. Each field is qualified with the host structure names of all levels to prevent naming conflicts with other host variables or fields. This is equivalent to the following method.

2. Fully qualified host variable names can be referenced in an SQL statement.

   ```
   EXEC SQL SELECT id, name, years, salary
        INTO :staff_record.id, :staff_record.name,
             :staff_record.info.years, :staff_record.info.salary
        FROM staff
        WHERE id = 10;
   ```

   References to field names must be fully qualified even if there are no other host variables with the same name. Qualified sub-structures can also be referenced. In the example above, `:staff_record.info` can be used to replace `:staff_record.info.years, :staff_record.info.salary`.

Since a reference to a host structure (first example) is equivalent to a comma-separated list of its fields, there are instances where this type of reference may lead to an error. For example:

```
EXEC SQL DELETE FROM :staff_record;
```

Here, the DELETE statement expects a single character-based host variable. By giving a host structure instead, the statement results in a precompile-time error:

```
SQL0087N  Host variable "staff_record" is a structure used where structure
references are not permitted.
```

Other uses of host structures, which may cause an SQL0087N error to occur, include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables and SQLDA references. Host structures with exactly one field are permitted in such situations, as are references to individual fields (second example).

Indicator Tables in C and C++

An indicator table is a collection of indicator variables to be used with a host structure. It must be declared as an array of short integers. For example:

```
short ind_tab[10];
```

The example above declares an indicator table with 10 elements. The following shows the way it can be used in an SQL statement:

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
     FROM staff
     WHERE id = 10;
```

The following lists each host structure field with its corresponding indicator variable in the table:

staff_record.id	ind_tab[0]
staff_record.name	ind_tab[1]
staff_record.info.years	ind_tab[2]
staff_record.info.salary	ind_tab[3]

Note: An indicator table element, for example `ind_tab[1]`, cannot be referenced individually in an SQL statement. The keyword INDICATOR is optional. The number of structure fields and indicators do not have to match; any extra indicators are unused, and any extra fields do not have indicators assigned to them.

A scalar indicator variable can also be used in the place of an indicator table to provide an indicator for the first field of the host structure. This is equivalent to having an indicator table with only 1 element. For example:

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

If an indicator table is specified along with a host variable instead of a host structure, only the first element of the indicator table, for example `ind_tab[0]`, will be used:

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

If an array of short integers is declared within a host structure:

```
struct tag
{
  short i[2];
} test_record;
```

The array will be expanded into its elements when `test_record` is referenced in an SQL statement making `:test_record` equivalent to `:test_record.i[0]`, `:test_record.i[1]`.

Null-terminated Strings in C and C++

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 in C and C++

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 FROM table;   /* 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 in C and C++

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;
    .
    .
};
```

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;
       }
};
```

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
"Example of Declaring Class Data Members as Host Variables" on page 620), 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;
}
```

Using Qualification and Member Operators in C and C++

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 in C and C++

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 "Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)" on page 606.)
- 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 "Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)" on page 606.)
- VARGRAPHIC structured form.

  VARGRAPHIC structured host variables have an SQLTYPE of 464/465 if their length is between 1 and 16 336 bytes. They have an SQLTYPE of 472/473 if their length is between 2000 and 16 350 bytes. (See "Syntax for Graphic Declaration (VARGRAPHIC Structured Form)" on page 607.)

Multi-byte Character Encoding in C and C++

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. Subroutines that conform to ANSI C and X/OPEN Portability Guide 4 (XPG4) are 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 in C and C++

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 and UCS-2 Code Set Considerations" on page 521 and refer to the *Administration Guide*.

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 in C and C++").

Note: When specifying the WCHARTYPE CONVERT option on a Windows platform, you should note that wchar_t on Windows platforms is Unicode. Therefore, if your C/C++ compiler's wchar_t is not Unicode, the wcstombs() function call may fail with SQLCODE -1421 (SQLSTATE=22504). If this happens, you can specify the WCHARTYPE NOCONVERT option, and explicitly call the wcstombs() and mbstowcs() functions from within your program.

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 in C and C++

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.

Refer to 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, or for dealing with GRAPHIC data in any application (including single-byte applications) connected to a UCS-2 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, or in UCS-2 if connected to a UCS-2 database, 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, or in UCS-2 if connected to a UCS-2 database, 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 and UCS-2 databases.
- Data stored in DBCLOB files through the use of DBCLOB file reference variables is stored in either DBCS format, or, in the case of UCS-2 databases, in UCS-2 format. Likewise, input data from DBCLOB files is retrieved either in DBCS format, or, in the case of UCS-2 databases, in UCS-2 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, and UCS-2 Considerations in C and C++

If your application code page is Japanese or Traditional Chinese EUC, or if your application connects to a UCS-2 database, 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 DB2 client converts graphic data from the wide character format to your application code page, 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 application code page identifier. When graphic data is retrieved from a database by a client, it is tagged with the UCS-2 code page identifier. The DB2 client converts the data from UCS-2 to the client application code page, 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 the application code page to UCS-2 and from UCS-2 to the application code page 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. For the client environments where `wchar_t` encoding is in two-byte Unicode, for example Windows NT or AIX version 4.3 and higher, you can use the NOCONVERT option and work directly with UCS-2. In such cases, your application should handle the difference between big-endian and little-endian architectures. With NOCONVERT option, DB2 Universal Database uses sqldbchar which is always two-byte big-endian.

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.

In general, although eucJP and eucTW store GRAPHIC data as UCS-2, the GRAPHIC data in these database is still non-ASCII eucJP or eucTW data. Specifically, any space padded to such GRAPHIC data is DBCS space (also known as ideographic space in UCS-2, U+3000). For a UCS-2 database, however, GRAPHIC data can contain any UCS-2 character, and space padding is done with UCS-2 space, U+0020. Keep this difference in mind when you code applications to retrieve UCS-2 data from a UCS-2 database versus UCS-2 data from eucJP and eucTW databases.

For general EUC application development guidelines, see "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 521.

Supported SQL Data Types in C and C++

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 30 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.

Note: There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 30. 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 sqlint16	16-bit signed integer
INTEGER (496 or 497)	long long int sqlint32[2]	32-bit signed integer
BIGINT (492 or 493)	long long long __int64 sqlint64[3]	64-bit signed integer
REAL[4] (480 or 481)	float	Single-precision floating point
DOUBLE[5] (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

VARCHAR(*n*)
(448 or 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		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*<=32 672	null-terminated variable-length character string **Note:** Assigned an SQL type of 460/461.

Table 30. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description

LONG VARCHAR
(456 or 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*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[6] (964 or 965)	sql type is clob_locator	Identifies CLOB entities residing on the server
CLOB file reference variable[6] (920 or 921)	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[6] (960 or 961)	sql type is blob_locator	Identifies BLOB entities on the server
BLOB file reference variable[6] (916 or 917)	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.
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

Table 30. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 336
		Non null-terminated varying double-byte character string with 2-byte string length indicator

	Alternately use sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336	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*]
	}
	
	16 337<=*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

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	1<=*n*<=16 336
		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*<=16 336	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*]
	}
	
	16 337<=*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[6] (968 or 969)	sql type is dbclob_locator	Identifies DBCLOB entities residing on the server

Table 30. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description
DBCLOB file reference variable[6] (924 or 925)	sql type is dbclob_file	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **SQL 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. For platform compatibility, use sqlint32. On 64-bit UNIX platforms, "long" is a 64 bit integer. On 64-bit Windows operating systems and 32-bit UNIX platforms "long" is a 32 bit integer.
3. For platform compatibility, use sqlint64. The DB2 Universal Database sqlsystm.h header file will type define sqlint64 as "__int64" on the Windows NT platform when using the Microsoft compiler, "long long" on 32-bit UNIX platforms, and "long" on 64 bit UNIX platforms.
4. 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).
5. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is
   - DOUBLE PRECISION
6. 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 */
        sqlint32  salary;                /* SQL type  496 */
        sqlint32  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[32695];
                 } 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  920 */
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  916 */
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  924 */
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 in C and C++

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.

C/C++ Types for Stored Procedures, Functions, and Methods

The following table lists the supported mappings between SQL data types and C data types for stored procedures, UDFs, and methods.

Table 31. SQL Data Types Mapped to C/C++ Declarations

SQL Column Type	C/C++ Data Type	SQL Column Type Description
SMALLINT (500 or 501)	sqlint16	16-bit signed integer
INTEGER (496 or 497)	sqlint32	32-bit signed integer
BIGINT (492 or 493)	sqlint64	64-bit signed integer
REAL (480 or 481)	float	Single-precision floating point
DOUBLE (480 or 481)	double	Double-precision floating point

Table 31. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
DECIMAL(*p*,*s*) (484 or 485)	Not supported.	To pass a decimal value, 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.
CHAR(*n*) (452 or 453)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length, null-terminated character string
CHAR(*n*) FOR BIT DATA (452 or 453)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length character string
VARCHAR(*n*) (448 or 449) (460 or 461)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=32 672	Null-terminated varying length string

VARCHAR(*n*) FOR BIT DATA
(448 or 449)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
		Not null-terminated varying length character string

LONG VARCHAR
(456 or 457)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		Not null-terminated varying length character string

CLOB(*n*)
(408 or 409)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		Not null-terminated varying length character string with 4-byte string length indicator

BLOB(*n*)
(404 or 405)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		Not null-terminated varying binary string with 4-byte string length indicator

DATE (384 or 385)	char[11]	Null-terminated character form
TIME (388 or 389)	char[9]	Null-terminated character form
TIMESTAMP (392 or 393)	char[27]	Null-terminated character form

Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.

Table 31. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
GRAPHIC(*n*) (468 or 469)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length, null-terminated double-byte character string
VARGRAPHIC(*n*) (400 or 401)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336	Not null-terminated, variable-length double-byte character string

LONG VARGRAPHIC
(472 or 473)
	struct {
	sqluint16 length;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		Not null-terminated, variable-length double-byte character string

DBCLOB(*n*)
(412 or 413)
	struct {
	sqluint32 length;
	sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
		Not null-terminated varying length character string with 4-byte string length indicator

SQLSTATE and SQLCODE Variables in C and C++

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]
  sqlint32  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 sqlint32 SQLCODE;
extern char     SQLSTATE[6];
```

Chapter 21. Programming in Java

Programming Considerations for Java

DB2 Universal Database implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ). This chapter provides an overview of JDBC and SQLJ programming, but focuses on the aspects specific to DB2. See the DB2 Universal Database Java Web site at `http://www.ibm.com/software/data/db2/java/` for links to the JDBC and SQLJ specifications.

Comparison of SQLJ to JDBC

The JDBC API allows you to write Java programs that make dynamic SQL calls to databases. SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. You must translate a SQLJ source file with the SQLJ translator before you can compile the resulting Java source code.

For information on building JDBC and SQLJ applications, refer to the *Application Building Guide*.

Advantages of Java Over Other Languages

Programming languages containing embedded SQL are called host languages. Java differs from the traditional host languages C, COBOL, and FORTRAN, in ways that significantly affect how it embeds SQL:

- SQLJ and JDBC are open standards, enabling you to easily port SQLJ or JDBC applications from other standards-compliant database systems to DB2 Universal Database.
- All Java types representing composite data, and data of varying sizes, have a distinguished value, `null`, which can be used to represent the SQL NULL state, giving Java programs an alternative to NULL indicators that are a fixture of other host languages.
- Java is designed to support programs that are automatically heterogeneously portable (also called ″super portable″ or simply ″downloadable″). Along with Java's type system of classes and interfaces, this feature enables component software. In particular, an SQLJ translator written in Java can call components that are specialized by database vendors in order to leverage existing database functions such as authorization, schema checking, type checking, transactional, and recovery capabilities, and to generate code optimized for specific databases.
- Java is designed for binary portability in heterogeneous networks, which promises to enable binary portability for database applications that use static SQL.

SQL Security in Java

By default, a JDBC program executes SQL statements with the privileges assigned to the person who runs the program. In contrast, an SQLJ program executes SQL statements with the privileges assigned to the person who created the database package.

Source and Output Files for Java

Source files have the following extensions:

.java Java source files, which require no precompiling. You can compile these files with the `javac` Java compiler included with your Java development environment.

.sqlj SQLJ source files, which require translation with the `sqlj` translator. The translator creates:

- one or more `.class` bytecode files
- one `.ser` profile file per connection context

The corresponding output files have the following extensions:

.class JDBC and SQLJ bytecode compiled files.

.ser SQLJ serialized profile files. You create packages in the database for each profile file with the `db2profc` utility.

For an example of how to compile and run an SQLJ program, see "Compiling and Running SQLJ Programs" on page 660.

Java Class Libraries

DB2 Universal Database provides class libraries for JDBC and SQLJ support, which you must provide in your CLASSPATH or include with your applets as follows:

db2java.zip
Provides the JDBC driver and JDBC and SQLJ support classes, including stored procedure and UDF support.

sqlj.zip
Provides the SQLJ translator class files.

runtime.zip
Provides Java run-time support for SQLJ applications and applets.

Java Packages

To use the class libraries included with DB2 in your own applications, you must include the appropriate `import` *`package`* statements at the top of your source files. You can use the following packages in your Java applications:

java.sql.*
The JDBC API included in your JDK. You must import this package in every JDBC and SQLJ program.

sqlj.runtime.*
SQLJ support included with every DB2 client. You must import this package in every SQLJ program.

sqlj.runtime.ref.*
SQLJ support included with every DB2 client. You must import this package in every SQLJ program.

Supported SQL Data Types in Java

Table 32 on page 640 shows the Java equivalent of each SQL data type, based on the JDBC specification for data type mappings. Note that some mappings depend on whether you use the JDBC version 1.2 or 2.0 driver. The JDBC driver converts the data exchanged between the application and the database using the following mapping schema. Use these mappings in your Java applications and your PARAMETER STYLE JAVA stored procedures and UDFs. For information on data type mappings for PARAMETER STYLE DB2GENERAL stored procedures and UDFs, see "Supported SQL Data Types" on page 770.

Note: There is no host variable support for the DATALINK data type in any of the programming languages supported by DB2.

Table 32. SQL Data Types Mapped to Java Declarations

SQL Column Type	Java Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short	16-bit, signed integer
INTEGER (496 or 497)	int	32-bit, signed integer
BIGINT (492 or 493)	long	64-bit, signed integer
REAL (480 or 481)	float	Single precision floating point
DOUBLE (480 or 481)	double	Double precision floating point
DECIMAL(*p*,*s*) (484 or 485)	java.math.BigDecimal	Packed decimal
CHAR(*n*) (452 or 453)	String	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	String	Variable-length character string
LONG VARCHAR (456 or 457)	String	Long variable-length character string
CHAR(*n*) FOR BIT DATA	byte[]	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) FOR BIT DATA	byte[]	Variable-length character string
LONG VARCHAR FOR BIT DATA	byte[]	Long variable-length character string
BLOB(*n*) (404 or 405)	JDBC 1.2: byte[] JDBC 2.0: java.sql.Blob	Large object variable-length binary string
CLOB(*n*) (408 or 409)	JDBC 1.2: String JDBC 2.0: java.sql.Clob	Large object variable-length character string
DBCLOB(*n*) (412 or 413)	JDBC 1.2: String JDBC 2.0: java.sql.Clob	Large object variable-length double-byte character string
DATE (384 or 385)	java.sql.Date	10-byte character string
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	26-byte character string

SQLSTATE and SQLCODE Values in Java

If an SQL error occurs, JDBC and SQLJ programs throw an `SQLException`. To retrieve the SQLSTATE, SQLCODE, or SQLMSG values for an instance of an `SQLException`, invoke the corresponding instance method as follows:

SQL return code	SQLException method
SQLCODE	`SQLException.getErrorCode()`
SQLMSG	`SQLException.getMessage()`
SQLSTATE	`SQLException.getSQLState()`

For example:

```
int sqlCode=0;      // Variable to hold SQLCODE
String sqlState="00000";  // Variable to hold SQLSTATE

try
{
  // JDBC statements may throw SQLExceptions
  stmt.executeQuery("Your JDBC statement here");

  // SQLJ statements may also throw SQLExeptions
  #sql {..... your SQLJ statement here ......};
}

/* Here's how you can check for SQLCODEs and SQLSTATE */

catch (SQLException e)
{
  sqlCode = e.getErrorCode() // Get SQLCODE
  sqlState = e.getSQLState() // Get SQLSTATE

  if (sqlCode == -190 || sqlState.equals("42837"))
  {
    // Your code here to handle SQLCODE -190 or SQLSTATE 42837
  }
  else
  {
    // Your code here to handle other errors
  }
  System.err.println(e.getMessage()) ; // Print the exception
  System.exit(1);         // Exit
}
```

Trace Facilities in Java

Both the CLI/ODBC/JDBC trace facility and the DB2 trace facility, `db2trc`, can be used to diagnose problems related to JDBC or SQLJ programs. Details on how to take the above traces are explained in the *Troubleshooting Guide*.

You can also install run-time call tracing capability into SQLJ programs. The utility operates on the profiles associated with a program. Suppose a program uses a profile called `App_SJProfile0`. To install call tracing into the program, use the command:

```
profdb App_SJProfile0.ser
```

The `profdb` utility uses the Java Virtual Machine to run the `main()` method of class `sqlj.runtime.profile.util.AuditorInstaller`. For more details on usage and options for the `AuditorInstaller` class, visit the DB2 Java Web site at `http://www.ibm.com/software/data/db2/java`.

Creating Java Applications and Applets

Whether your application or applet uses JDBC or SQLJ, you need to familiarize yourself with the JDBC specification, which is available from Sun Microsystems. See the DB2 Java Web site at `http://www.ibm.com/software/data/db2/java/` for links to JDBC and SQLJ resources. 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 (see "JDBC 2.0" on page 649). If you plan to create UDFs or stored procedures in Java, see "Creating and Using Java User-Defined Functions" on page 420 and "Java Stored Procedures and UDFs" on page 668, as there are considerations that are different for Java than for other languages.

To build and run JDBC and SQLJ applications and applets, you must set up your operating system environment according to the instructions in the *Application Building Guide*.

How Does It Work?

DB2's Java enablement has three independent components:

- Support for client applications and applets written in Java using JDBC to access DB2 (see "JDBC Programming" on page 644)
- Precompile and binding support for client applications and applets written in Java using SQLJ to access DB2 (see "SQLJ Programming" on page 651)
- Support for Java UDFs and stored procedures on the server (see "Stored Procedures and UDFs in Java" on page 663)

Application Support in Java: Figure 21 on page 643 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. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server.

SQLJ applications use this JDBC support, and in addition require the SQLJ run-time classes to authenticate and execute any SQL packages that were bound to the database at the precompiling and binding stage.

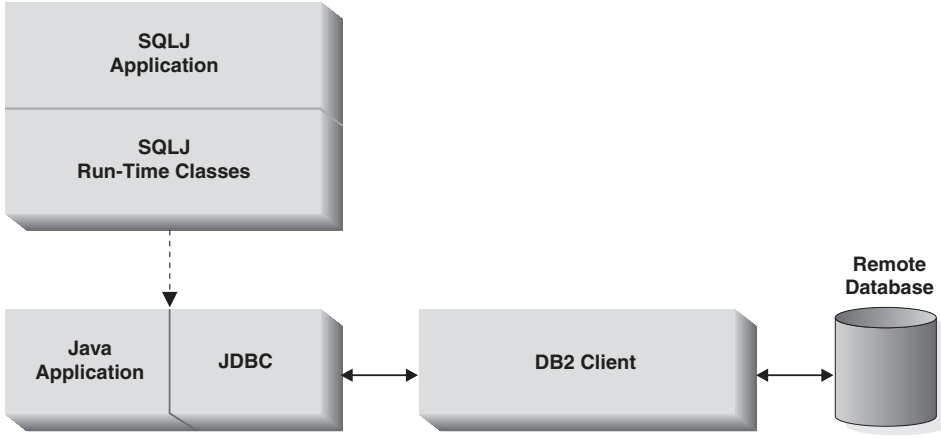


Figure 21. DB2's Java Application Implementation

Applet Support in Java: Figure 22 on page 644 illustrates how the JDBC *applet driver*, also known as the *net driver*, works. The driver consists of a JDBC client and a JDBC server, `db2jd`. The JDBC client driver is loaded on the Web browser along with the applet. When the applet requests a connection to a DB2 database, 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.

SQLJ applets add the SQLJ client driver on top of the JDBC client driver, but otherwise work the same as JDBC applets.

For information on starting the DB2 JDBC server, refer to the `db2jstrt` command in the *Command Reference*.

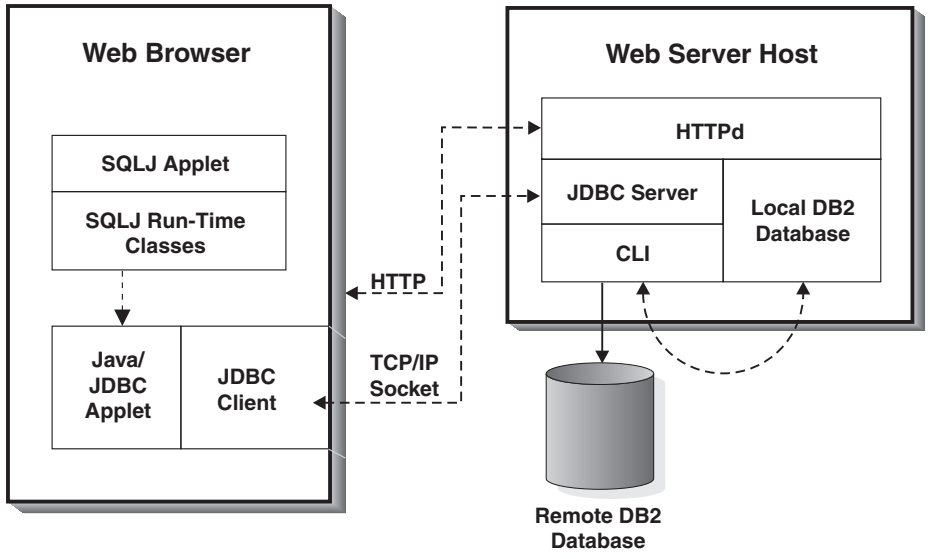


Figure 22. DB2 Java Applet Implementation

JDBC Programming

Both applications and applets typically perform the following tasks:

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 the JDBC specification and using the `db2` subprotocol. Applets require you to provide the user ID, password, host name, and the port number for the applet server. Applications implicitly use the default value for user ID and password from the DB2 client catalog, unless you explicitly specify alternate 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.

How the DB2Appl Program Works

The following sample program, DB2Appl.java, demonstrates how to code a JDBC program for DB2.

1. **Import the JDBC package.** Every JDBC and SQLJ program must import the JDBC package.
2. **Declare a Connection object.** The Connection object establishes and manages the database connection.
3. **Set database URL variable.** The DB2 application driver accepts URLs that take the form of `jdbc:db2:>database-name<`.
4. **Connect to database.** The `DriverManager.getConnection()` method is most often used with the following parameters:

   `getConnection(String url)`
   Establish a connection to the database specified by *url* with the default user ID and password.

   `getConnection(String url, String userid, String password)`
   Establish a connection to the database specified by *url* with the values for user ID and password specified by *userid* and *passwd* respectively.
5. **Create a Statement object.** Statement objects send SQL statements to the database.
6. **Execute an SQL SELECT statement.** Use the `executeQuery()` method for SQL statements, like SELECT statements, that return a single result set. Assign the result to a ResultSet object.
7. **Retrieve rows from the ResultSet.** The ResultSet object allows you to treat a result set like a cursor in host language embedded SQL. The `ResultSet.next()` method moves the cursor to the next row and returns a `boolean false` if the final row of the result set has been reached. Restrictions on result set processing depend on the level of the JDBC API that is enabled through the database manager configuration parameters.
   - The JDBC 2.0 API allows you to scroll backwards and forwards through a result set.
   - The JDBC 1.2 API restricts you to scrolling forward through a result set with the `ResultSet.next()` method.
8. **Return the value of a column.** The `ResultSet.getString(n)` returns the value of the n^{th} column as a String object.
9. **Execute an SQL UPDATE statement.** Use the `executeUpdate()` method for SQL UPDATE statements. The method returns the number of rows updated as an int value.

JDBC Example: DB2Appl.java

```
import java.sql.*; 1

class DB2Appl {

   static {
      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
      } catch (Exception e) {
         System.out.println(e);
      }
   }

   public static void main(String argv[]) {
      Connection con = null; 2

      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample"; 3

      try {
         if (argv.length == 0) {
            // connect with default id/password
            con = DriverManager.getConnection(url);
            }
         else if (argv.length == 2) {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd); 4
            }
         else {
            System.out.println("Usage: java DB2Appl [username password]");
            System.exit(0);
         }

         // retrieve data from the database
         System.out.println("Retrieve some data from the database...");
         Statement stmt = con.createStatement(); 5
         ResultSet rs = stmt.executeQuery("SELECT * from employee"); 6

         System.out.println("Received results:");

         // display the result set
         // rs.next() returns false when there are no more rows
         while (rs.next()) { 7
            String a = rs.getString(1); 8
            String str = rs.getString(2);

            System.out.print(" empno= " + a);
            System.out.print(" firstname= " + str);
            System.out.print("");
         }

         rs.close();
```

```
            stmt.close();

            // update the database
            System.out.println("Update the database... ");
            stmt = con.createStatement();
            int rowsUpdated = stmt.executeUpdate("UPDATE employee
           SET firstnme = 'SHILI' where empno = '000010'"); 9

            System.out.print("Changed "+rowsUpdated);

            if (1 == rowsUpdated)
               System.out.println(" row.");
            else
               System.out.println(" rows.");

            stmt.close();
            con.close();
         } catch( Exception e ) {
            System.out.println(e);
         }
      }
   }
```

Distributing a JDBC Application

Distribute your JDBC application as you would any other Java application. As the application uses the DB2 client to communicate with the DB2 server, you have no special security concerns; authority verification is performed by the DB2 client.

To run your application on a client machine, you must install on that machine:

- A Java Virtual Machine (JVM), which you need to run any Java code
- A DB2 client, which also includes the DB2 JDBC driver

To build your application, you must also install the JDK for your operating system. For information on setting up your Java environment, building DB2 Java applications, and running DB2 Java applications, refer to the *Application Building Guide*.

Distributing and Running a JDBC Applet

Like other Java applets, you distribute your JDBC applet over the network (intranet or Internet). Typically you would embed 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 following `<APPLET>` tag:

```
<applet code="DB2Applt.class" width=325 height=275 archive="db2java.zip">
   <param name="server" value="webhost">
   <param name="port" value="6789">
</applet>
```

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 instructs your browser to download the Java applet and the db2java.zip class library, which includes the DB2 JDBC driver implemented by the `COM.ibm.db2.jdbc.net` class. 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 running on the Web server.

Note: To ensure that the Web browser downloads db2java.zip from the server, ensure that the CLASSPATH environment variable on the client does *not* include db2java.zip. Your applet may not function correctly if the client uses a local version of db2java.zip.

For information on building and distributing Java applets, refer to the *Application Building Guide*.

Connecting to the JDBC Applet Server

It is essential that the db2java.zip file used by the Java applet be at the same FixPak level as the JDBC applet server. Under normal circumstances, db2java.zip is loaded from the Web Server where the JDBC applet server is running, as shown in Figure 22 on page 644. This ensures a match. If, however, your configuration has the Java applet loading db2java.zip from a different location, a mismatch can occur. Prior to DB2 Version 7.1 FixPak 2, this could lead to unexpected failures. As of DB2 Version 7.1 FixPak 2, matching FixPak levels between the two files is strictly enforced at connection time. If a mismatch is detected, the connection is rejected, and the client receives one of the following exceptions:

- If db2java.zip is at DB2 Version 7.1 FixPak 2 or later:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0621E  Unsupported JDBC server configuration.
```

- If db2java.zip is prior to DB2 Version 7.1 FixPak 2:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0601E  Invalid statement handle or statement is closed.
SQLSTATE=S1000
```

If a mismatch occurs, the JDBC applet server logs one of the following messages in the jdbcerr.log file:

- If the JDBC applet server is at DB2 Version 7.1 FixPak 2 or later:

```
jdbcFSQLConnect: JDBC Applet Server and client (db2java.zip)
versions do not match. Unable to proceed with connection., einfo= -111
```

- If the JDBC applet server is prior to DB2 Version 7.1 FixPak 2:

```
jdbcServiceConnection(): Invalid Request Received., einfo= 0
```

JDBC 2.0

JDBC 2.0 is the latest version of JDBC from Sun. This version of JDBC has two defined parts: the **core API**, and the **Optional Package API**. For information on the JDBC specification, see the DB2 Universal Database Java Web site at `http://www.ibm.com/software/data/db2/java/`.

For information on installing the JDBC 2.0 drivers for your operating system, refer to the *Application Building Guide*.

JDBC 2.0 Core API Support

The DB2 JDBC 2.0 driver supports the JDBC 2.0 core API, however, it does not support all of the features defined in the specification. The DB2 JDBC 2.0 driver supports the following features of the JDBC 2.0 core API:

- Scrollable insensitive ResultSet
- Batch updates for `java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement`
- `java.sql.Blob` support
- `java.sql.Clob` support

The DB2 JDBC 2.0 driver does not support the following features:

- Updatable Scrollable ResultSet
- New SQL types (Array, Ref, Distinct, Java Object, Struct)
- Customized SQL type mapping
- java.sql.Blob or java.sql.Clob in Java stored procedures, UDFs or methods.
- Scrollable sensitive ResultSets (scroll type of ResultSet.TYPE_SCROLL_SENSITIVE)
- ResultSet.setFetchDirection(int) (ignored, does not throw Exception)
- ResultSet.setFetchSize(int) (ignored, does not throw Exception)
- Statement.setFetchSize(int) (ignored, does not throw Exception)
- ResultSet.getTime(int, Calendar)
- ResultSet.getTimestamp(int, Calendar)
- CallableStatement.getClob()
- CallableStatement.getBlob()

JDBC 2.0 Optional Package API Support

The DB2 JDBC 2.0 driver supports the following features of the JDBC 2.0 Optional Package API:

Java Naming and Directory Interface (JNDI) for Naming Databases: DB2 provides the following support for the Javing Naming and Directory Interface (JNDI) for naming databases:

javax.naming.Context

This interface is implemented by `COM.ibm.db2.jndi.DB2Context`, which handles the storage and retrieval of DataSource objects. In order to support persistent associations of logical data source names to physical database information, such as database names, these associations are saved in a file named .db2.jndi. For an application, the file resides (or is created if none exists) in the directory specified by the USER.HOME environment variable. For an applet, you must create this file in the root directory of the web server to facilitate the `lookup()` operation. Applets do not support the `bind()`, `rebind()`, `unbind()` and `rename()` methods of this class. Only applications can bind DataSource objects to JNDI.

javax.sql.Datasource

This interface is implemented by `COM.ibm.db2.jdbc.DB2DataSource`. You can save an object of this class in any implementation of `javax.naming.Context`. This class also makes use of connection pooling support.

`DB2DataSource` supports the following methods:

- public void setDatabaseName(String databaseName)
- public void setServerName(String serverName)
- public void setPortNumber(int portNumber)

javax.naming.InitialContextFactory

This interface is implemented by `COM.ibm.db2.jndi.DB2InitialContextFactory`, which creates an instance of `DB2Context`. Applications automatically set the value of the JAVA.NAMING.FACTORY.INITIAL environment variable to `COM.ibm.db2.jndi.DB2InitialContextFactory` To use this class in an applet, call InitialContext() using the following syntax:

```
Hashtable env = new Hashtable( 5 );
env.put( "java.naming.factory.initial",
         "COM.ibm.db2.jndi.DB2InitialContextFactory" );
Context ctx = new InitialContext( env );
```

Connection Pooling: `DB2ConnectionPoolDataSource` and `DB2PooledConnection` provide the hooks necessary for you to implement your own connection pooling module, as follows:

javax.sql.ConnectionPoolDataSource

This interface is implemented by `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`, and is a factory of `COM.ibm.db2.jdbc.DB2PooledConnection` objects.

javax.sql.PooledConnection

This interface is implemented by `COM.ibm.db2.jdbc.DB2PooledConnection`.

Java Transaction APIs (JTA): DB2 supports the Java Transaction APIs (JTA) through the DB2 JDBC application driver. DB2 does not provide JTA support with the DB2 JDBC net driver.

javax.sql.XAConnection
This interface is implemented by `COM.ibm.db2.jdbc.DB2XAConnection`.

javax.sql.XADataSource
This interface is implemented by `COM.ibm.db2.jdbc.DB2XADataSource`, and is a factory of `COM.ibm.db2.jdbc.DB2PooledConnection` objects.

javax.transactions.xa.XAResource
This interface is implemented by `COM.ibm.db2.jdbc.app.DBXAResource`.

javax.transactions.xa.Xid
This interface is implemented by `COM.ibm.db2.jdbc.DB2Xid`.

JDBC 2.0 Compatibility

This version of the specification is backward compatible with the previous version (1.2). However, the DB2 JDBC 1.2 driver supports LOB types as an extension of the JDBC 1.2 specification, and this extension is not part of the new specification's backward compatibility. This means that existing JDBC applications that rely on the LOB support of the JDBC 1.2 driver may not work with the new driver. For information on the DB2 JDBC 1.2 driver support for LOBs and graphic types, see "Using LOBs and Graphical Objects With JDBC 1.2" on page 672. To fix the problem, consider modifying the application to take advantage of the LOB support offered by the JDBC 2.0 driver.

Note: You cannot use the DB2 JDBC 2.0 driver support for LOB and graphic types in stored procedures or UDFs. To use LOB or graphic types in stored procedures or UDFs, you must use the JDBC 1.2 driver support.

SQLJ Programming

DB2 SQLJ support is based on the SQLJ ANSI standard. Refer to the DB2 Java Web site at `http://www.ibm.com/software/data/db2/java` for a pointer to the ANSI Web site and other SQLJ resources. This chapter contains an overview of SQLJ programming and information that is specific to DB2 SQLJ support.

The following kinds of SQL constructs may appear in SQLJ programs:

- Queries; for example, SELECT statements and expressions.
- SQL Data Change Statements (DML); for example, INSERT, UPDATE, DELETE.
- Data Statements; for example, FETCH, SELECT..INTO.
- Transaction Control; for example, COMMIT, ROLLBACK, etc.

- Data Definition Language (DDL, also known as Schema Manipulation Language); for example, CREATE, DROP, ALTER.
- Calls to stored procedures; for example, CALL MYPROC(:x, :y, :z)
- Invocations of functions; for example, VALUES(MYFUN(:x))

DB2 SQLJ Support

DB2 SQLJ support is provided by the DB2 Application Development Client. Along with the JDBC support provided by the DB2 client, DB2 SQLJ support allows you to create, build, and run embedded SQL for Java applications, applets, stored procedures and user-defined functions (UDFs). These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

The SQLJ support provided by the DB2 Application Development Client includes:

- The SQLJ translator, `sqlj`, which replaces embedded SQL statements in the SQLJ program with Java source statements and generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the `sqllib/java/sqlj.zip` file.
- The SQLJ run-time classes, available in `sqllib/java/runtime.zip`.
- The DB2 SQLJ profile customizer, `db2profc`, which precompiles the SQL statements stored in the generated profile and generates a package in the DB2 database.
- The DB2 SQLJ profile printer, `db2profp`, which prints the contents of a DB2 customized profile in plain text.
- The SQLJ profile auditor installer, `profdb`, which installs (or uninstalls) debugging class-auditors into an existing set of binary profiles. Once installed, all `RTStatement` and `RTResultSet` calls made during application run time are logged to a file (or standard output), which can then be inspected to verify expected behavior and trace errors. Note that only those calls made to the underlying `RTStatement` and `RTResultSet` call interface at run time are audited.
- The SQLJ profile conversion tool, `profconv`, which converts a serialized profile instance to class bytecode format. Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. As a work-around, you need to run this utility to perform the conversion.

For more information on the `db2profc` and `db2profp` commands, refer to the *Command Reference*. For more information on the SQLJ run-time classes, refer to the DB2 Java Web site at `http://www.ibm.com/software/data/db2/java`.

DB2 SQLJ Restrictions

When you create DB2 applications with SQLJ, you should be aware of the following restrictions:

- DB2 SQLJ support adheres to standard DB2 Universal Database restrictions on issuing SQL statements.
- A positioned UPDATE and DELETE statement is not a valid sub-statement in a Compound SQL statement.
- The precompile option "DATETIME" is not supported. Only the date and time formats of the International Standards Organization are supported.
- The precompile option "PACKAGE USING package-name" specifies the name of the package that is to be generated by the translator. If a name is not entered, the name of the profile (minus extension and folded to uppercase) is used. Maximum length is 8 characters. Since the SQLJ profile name has the suffix _SJProfile*N*, where *N* is the profile key number, the profile name will always be longer than 8 characters. The default package name will be constructed by concatenating the first (8 - *pfKeyNumLen*) characters of the profile number and the profile key number, where *pfKeyNumLen* is the length of the profile key number in the profile name. If the length of the profile key number is longer than 7, the last 7 digits will be used without any warnings. For example:

```
profile name                    default package name
--------------------            --------------------
App_SJProfile1                  App_SJP1
App_SJProfile123                App_S123
App_SJProfile1234567            A1234567
App_SJProfile12345678           A2345678
```

- When a `java.math.BigDecimal` host variable is used, the precision and scale of the host variable is not available during the translation of the application. If the precision and scale of the decimal host variable is not obvious from the context of the statement in which it is used, the precision and scale can be specified using a CAST.
- A Java variable with type `java.math.BigInteger` cannot be used as a host variable in an SQL statement.

Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. You will get the following error message when trying to load the applet Applt in those browsers:

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

As a work-around, there is a utility which converts a serialized profile into a profile stored in Java class format. The utility is a Java class called `sqlj.runtime.profile.util.SerProfileToClass`. It takes a serialized profile resource file as input and produces a Java class containing the profile as output. Your profile can be converted using the following command:

```
profconv Applt_SJProfile0.ser

  or

java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

The class Applt_SJProfile0.class is created as a result. Replace all profiles in .ser format used by the applet with profiles in .class format.

For an SQLJ applet, you need both db2java.zip and runtime.zip files. If you choose not to package all your applet classes, classes in db2java.zip and runtime.zip into a single Jar file, put both db2java.zip and runtime.zip (separated by a comma) into the archive parameter in the "applet" tag. For those browsers that do not support multiple zip files in the archive tag, specify db2java.zip in the archive tag, and unzip runtime.zip with your applet classes in a working directory that is accessible to your web browser.

Embedding SQL Statements in Java

Static SQL statements in SQLJ appear in *SQLJ clauses*. SQLJ clauses are the mechanism by which SQL statements in Java programs are communicated to the database.

The SQLJ translator recognizes SQLJ clauses and SQL statements because of their structure, as follows:

- SQLJ clauses begin with the token #sql
- SQLJ clauses end with a semicolon

The simplest SQLJ clauses are *executable clauses* and consist of the token #sql followed by an SQL statement enclosed in braces. For example, the following SQLJ clause may appear wherever a Java statement may legally appear. Its purpose is to delete all rows in the table named TAB:

```
#sql { DELETE FROM TAB };
```

In an SQLJ executable clause, the tokens that appear inside the braces are SQL tokens, except for the host variables. All host variables are distinguished by the colon character so the translator can identify them. SQL tokens never occur outside the braces of an SQLJ executable clause. For example, the following Java method inserts its arguments into an SQL table. The method body consists of an SQLJ executable clause containing the host variables *x*, *y*, and *z*:

```
void m (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

In general, SQL tokens are case insensitive (except for identifiers delimited by double quotation marks), and can be written in upper, lower, or mixed case. Java tokens, however, are case sensitive. For clarity in examples, case insensitive SQL tokens are uppercase, and Java tokens are lowercase or mixed case. Throughout this chapter, the lowercase null is used to represent the Java "null" value, and the uppercase NULL to represent the SQL null value.

Declaring Iterator Behavior in SQLJ

Unlike SQL statements that retrieve data from a table, applications that perform positioned UPDATE and DELETE operations, or that use iterators with `holdability` or `returnability` attributes, require two Java source files. Declare the iterator as `public` in one source file, appending the `with` and `implements` clause as appropriate.

To set the value of the `holdability` or `returnability` attribute, you must declare the iterator using the `with` clause for the corresponding attribute. The following example sets the `holdability` attribute to `true` for the iterator `WithHoldCurs`:

```
#sql public iterator WithHoldCurs with (holdability=true) (String EmpNo);
```

Iterators that perform positioned updates require an `implements` clause that implements the `sqlj.runtime.ForUpdate` interface. For example, suppose that you declare iterator DelByName like this in `file1.sqlj`:

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

You can then use the translated and compiled iterator in a different source file. To use the iterator:

1. Declare an instance of the generated iterator class
2. Assign the SELECT statement for the positioned UPDATE or DELETE to the iterator instance
3. Execute positioned UPDATE or DELETE statements using the iterator

To use DelByName for a positioned DELETE in `file2.sqlj`, execute statements like those in "Deleting Rows Using a Positioned Iterator".

```
{
   DelByName deliter; // Declare object of DelByName class
   String enum;
1  #sql deliter = { SELECT EMPNO FROM EMP WHERE WORKDEPT='D11'};
   while (deliter.next())
   {
2     enum = deliter.EmpNo(); // Get value from result table
3     #sql { DELETE WHERE CURRENT OF :deliter };
      // Delete row where cursor is positioned
   }
}
```

Notes:

1. **1** This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable *deliter*.
2. **2** This statement positions the iterator to the next row to be deleted.
3. **3** This SQLJ clause performs the positioned DELETE.

SQLJ Example: App.sqlj

The following example SQLJ application, App.sqlj, uses static SQL to retrieve and update data from the EMPLOYEE table of the DB2 `sample` database.

1. **Declare iterators.** This section declares two types of iterators:

   App_Cursor1
   Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns).

   App_Cursor2
   Declares column data types, and returns the values of the columns by column position (Positional binding to columns).

2. **Initialize the iterator.** The iterator object `cursor1` is initialized using the result of a query. The query stores the result in `cursor1`.
3. **Advance the iterator to the next row.** The `cursor1.next()` method returns a Boolean `false` if there are no more rows to retrieve.
4. **Move the data.** The named accessor method `empno()` returns the value of the column named `empno` on the current row. The named accessor method `firstnme()` returns the value of the column named `firstnme` on the current row.
5. **SELECT data into a host variable.** The SELECT statement passes the number of rows in the table into the host variable *count1*.
6. **Initialize the iterator.** The iterator object *cursor2* is initialized using the result of a query. The query stores the result in *cursor2*.
7. **Retrieve the data.** The FETCH statement returns the current value of the first column declared in the ByPos cursor from the result table into the host variable *str2*.
8. **Check the success of a FETCH..INTO statement.** The `endFetch()` method returns a Boolean `true` if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed. The `endFetch()` method returns `false` if the last attempt to fetch a row was successful. DB2 attempts to fetch a row when the `next()` method is called. A FETCH...INTO statement implicitly calls the `next()` method.
9. **Close the iterators.** The `close()` method releases any resources held by the iterators. You should explicitly close iterators to ensure that system resources are released in a timely fashion.

JDBC Example: App.sqlj:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; 1
#sql iterator App_Cursor2 (String) ;
```

```
class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
         System.exit(1);
       }
       DefaultContext.setDefaultContext(ctx);
     }
```

```
    // retrieve data from the database
    System.out.println("Retrieve some data from the database.");
    #sql cursor1 = {SELECT empno, firstnme FROM employee}; 2

    // display the result set
    // cursor1.next() returns false when there are no more rows
    System.out.println("Received results:");
    while (cursor1.next()) 3
    {
      str1 = cursor1.empno(); 4
      str2 = cursor1.firstnme();

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.print ("");
    }
    cursor1.close(); 9

    // retrieve number of employee from the database
    #sql { SELECT count(*) into :count1 FROM employee }; 5
    if (1 == count1)
      System.out.println ("There is 1 row in employee table");
    else
      System.out.println ("There are " + count1
                          + " rows in employee table");

    // update the database
    System.out.println("Update the database. ");
    #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

    // retrieve the updated data from the database
    System.out.println("Retrieve the updated data from the database.");
    str1 = "000010";
    #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; 6

    // display the result set
    // cursor2.next() returns false when there are no more rows
    System.out.println("Received results:");
    while (true)
    {
      #sql { FETCH :cursor2 INTO :str2 }; 7
      if (cursor2.endFetch()) break; 8

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.print ("");
    }
    cursor2.close(); 9

    // rollback the update
    System.out.println("Rollback the update.");
    #sql { ROLLBACK work };
    System.out.println("Rollback done.");
  }
```

```
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

Host Variables in Java

Arguments to embedded SQL statements are passed through *host variables*, which are variables of the host language that appear in the SQL statement. Host variables have up to three parts:

- A colon prefix, `:`.
- An optional parameter mode identifier: IN, OUT, or INOUT.
- A Java host variable that is a Java identifier for a parameter, variable, or field.

The evaluation of a Java identifier does not have side effects in a Java program, so it may appear multiple times in the Java code generated to replace an SQLJ clause.

The following query contains the host variable, *:x*, which is the Java variable, field, or parameter *x* visible in the scope containing the query:

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

All host variables specified in compound SQL are input host variables by default. You have to specify the parameter mode identifier OUT or INOUT before the host variable in order to mark it as an output host variable. For example:

```
#sql {begin compound atomic static
     select count(*) into :OUT count1 from employee;
     end compound}
```

Calls to Stored Procedures and Functions in SQLJ

Databases may contain *stored procedures*, *user-defined functions*, and *user-defined methods*. Stored procedures, user-defined functions, and user-defined methods are named schema objects that execute in the database. An SQLJ executable clause appearing as a Java statement may call a stored procedure by means of a CALL statement like the following:

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

Stored procedures may have IN, OUT, or INOUT parameters. In the above case, the value of host variable *myarg* is changed by the execution of that clause. An SQLJ executable clause may call a function by means of the SQL `VALUES` construct. For example, assume a function `F` that returns an integer. The following example illustrates a call to that function that then assigns its result to Java local variable *x*:

```
{
    int x;
    #sql x = { VALUES( F(34) ) };
}
```

Compiling and Running SQLJ Programs

To run an SQLJ program with program name `MyClass`, do the following:

1. Translate the Java source code with Embedded SQL to generate the Java source code `MyClass.java` and profiles `MyClass_SJProfile0.ser`, `MyClass_SJProfile1.ser`, ... (one profile for each connection context):

   ```
   sqlj MyClass.sqlj
   ```

   When you use the `SQLJ` translator without specifying an `sqlj.properties` file, the translator uses the following values:

   ```
   sqlj.url=jdbc:db2:sample
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   If you do specify an `sqlj.properties` file, make sure the following options are set:

   ```
   sqlj.url=jdbc:db2:dbname
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   where *dbname* is the name of the database. You can also specify these options on the command line. For example, to specify the database `mydata` when translating `MyClass`, you can issue the following command:

   ```
   sqlj -url=jdbc:db2:mydata MyClass.sqlj
   ```

   Note that the SQLJ translator automatically compiles the translated source code into class files, unless you explicitly turn off the compile option with the `-compile=false` clause.

2. Install DB2 SQLJ Customizers on generated profiles and create the DB2 packages in the DB2 database *dbname*:

   ```
   db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
     -prepoptions="bindfile using MyClass0.bnd package using MyClass0"
     MyClass_SJProfile0.ser
   db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
     -prepoptions="bindfile using MyClass1.bnd package using MyClass1"
     MyClass_SJProfile1.ser
   ...
   ```

3. Execute the SQLJ program:

   ```
   java MyClass
   ```

The translator generates the SQL syntax for the database for which the SQLJ profile is customized. For example,

```
i = { VALUES (  F(:x) ) };
```

is translated by the SQLJ translator and stored as

```
? = VALUES (F (?))
```

in the generated profile. When connecting to a DB2 Universal Database database, DB2 will customize the VALUE statement into:

```
VALUES(F(?)) INTO ?
```

but when connecting to a DB2 Universal Database for OS/390 database, DB2 customizes the VALUE statement into:

```
SELECT F(?) INTO ? FROM SYSIBM.SYSDUMMY1
```

If you run the DB2 SQLJ profile customizer, `db2profc`, against a DB2 Universal Database database and generate a bind file, you cannot use that bind file to bind up to a DB2 for OS/390 database when there is a `VALUES` clause in the bind file. This also applies to generating a bind file against a DB2 for OS/390 database and trying to bind with it to a DB2 Universal Database database.

For detailed information on building and running DB2 SQLJ programs, refer to the *Application Building Guide*.

SQLJ Translator Options

The SQLJ translator supports the same precompile options as the DB2 PRECOMPILE command, with the following exceptions:

```
CONNECT
DISCONNECT
DYNAMICRULES
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
SQLFLAG
SQLRULES
SYNCPOINT
TARGET
WCHARTYPE
```

To print the content of the profiles generated by the SQLJ translator in plain text, use the `profp` utility as follows:

```
profp MyClass_SJProfile0.ser
profp MyClass_SJProfile1.ser
  ...
```

To print the content of the DB2 customized version of the profile in plain text, use the `db2profp` utility as follows, where *dbname* is the name of the database:

```
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile0.ser
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile1.ser
    ...
```

Stored Procedures and UDFs in Java

You can create and use stored procedures and UDFs in Java just like you can for other programming languages. There are some programming considerations (as discussed in "Function Definitions in Java" on page 670) that you need to know when you write your Java code. You also need to *register* your stored procedure and UDFs. For information on how to register your stored procedure, see "Chapter 7. Stored Procedures" on page 193. For information on how to register your UDF, refer to the CREATE FUNCTION statement in the *SQL Reference*.

To run your UDFs and stored procedures on the server, DB2 calls the JVM. Ensure that the appropriate Java Development Kit (JDK) or Java Runtime Environment is installed and configured on your DB2 server before starting up the database.

The runtime libraries for the JVM must be available in the system search paths (PATH or LIBPATH or LD_LIBRARY_PATH, and CLASSPATH). For more information on setting up the Java environment, refer to the *Application Building Guide*.

DB2 loads or starts the JVM on the first call to a Java UDF or stored procedure. For NOT FENCED UDFs and stored procedures, DB2 loads one JVM per database instance, and runs it inside the address space of the database engine to improve performance. For FENCED UDFs, DB2 uses a distinct JVM inside the `db2udf` process; similarly, FENCED stored procedures use a distinct JVM inside the `db2dari` process. In all cases, the JVM stays loaded until the embedding process ends.

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.

You can study the Java stored procedure samples that are provided in the `sqllib/samples/java` directory. For a list of the sample programs included with DB2, see "Appendix B. Sample Programs" on page 743.

Remember that all Java class files that you use to implement a stored procedure or UDF must reside in either a JAR file you have installed in the database, or in the correct stored procedure or UDF path for your operating system as discussed in "Where to Put Java Classes" on page 664.

Note: On a mixed code page database server, Java user-defined functions and stored procedures cannot use CLOB type arguments, because random access on character boundaries on large mixed code page strings has not yet been implemented. Full support for all LOB types is intended

for SBCS databases. For mixed databases, support is intended for the BLOB and the DBCLOB types. As a workaround, applications running on a mixed database system should convert CLOB arguments to DBCLOB, LONG VARGRAPHIC, or LONG VARCHAR types. For UDFs, this can be done with the CAST operator.

Where to Put Java Classes

You can choose to use individual Java class files for your stored procedures and UDFs, or collect the class files into JAR files and install the JAR file in the database. If you decide to use JAR files, refer to "Java Stored Procedures and UDFs" on page 668 for further instructions.

Note: If you update or replace Java routine class files, you must issue a CALL SQLJ.REFRESH_CLASSES() statement to enable DB2 to load the updated classes. For more information on the CALL SQLJ.REFRESH_CLASSES() statement, refer to "Updating Java Classes for Routines" on page 665.

To enable DB2 to find and use your stored procedures and UDFs, you must store the corresponding class files in the *function directory*, which is a directory defined for your operating system as follows:

Unix operating systems

`sqllib/function`

OS/2 or Windows 32-bit operating systems

`instance_name``\function`, where *instance_name* represents the value of the *DB2INSTPROF* instance-specific registry setting.

For example, the function directory for a Windows NT server with DB2 installed in the `C:\sqllib` directory, and with no specified *DB2INSTPROF* registry setting, is:

```
C:\sqllib\function
```

If you choose to use individual class files, you must store the class files in the appropriate directory for your operating system. If you declare a class to be part of a Java package, create the corresponding subdirectories in the function directory and place the files in the corresponding subdirectory. For example, if you create a class `ibm.tests.test1` for a Linux system, store the corresponding Java bytecode file (named test1.class) in sqllib/function/ibm/tests.

The JVM that DB2 invokes uses the CLASSPATH environment variable to locate Java files. DB2 adds the function directory and sqllib/java/db2java.zip to the front of your CLASSPATH setting.

To set your environment so that the JVM can find the Java class files, you 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. For more information on configuration parameters, refer to the *Administration Guide*.

Updating Java Classes for Routines

When you update Java routine classes, you must also issue a CALL SQLJ.REFRESH_CLASSES() statement to force DB2 to load the new classes. If you do not issue the CALL SQLJ.REFRESH_CLASSES() statement after you update Java routine classes, DB2 continues to use the previous versions of the classes. The CALL SQLJ.REFRESH_CLASSES() statement only applies to FENCED routines. DB2 refreshes the classes when a COMMIT or ROLLBACK occurs.

Note: You cannot update NOT FENCED routines without stopping and restarting the database manager.

Debugging Stored Procedures in Java

DB2 provides the capability to interactively debug a stored procedure written in JDBC when the stored procedure executes on an AIX or Windows NT server. The easiest way to invoke debugging is through the DB2 Stored Procedure Builder. See the online help for the Stored Procedure Builder for more information about how to do this.

This section includes the following topics:

- Preparing to debug
- Populating the debug table
- Invoking the debugger

Preparing to Debug

1. Compile the stored procedure in debug mode according to your JDK documentation.
2. **Prepare the server**.
   - If the source code is stored on the server, set the CLASSPATH environment variable to include the Java source code directory or store the source code in the function directory, as defined in "Where to Put Java Classes" on page 664.
   - Use the `db2set` command to enable debugging for your instance:

     ```
     db2set DB2ROUTINE_DEBUG=ON
     ```
3. Set the client environment variables.
   - If the source code is stored on the client, set the DB2_DBG_PATH environment variable to the directory which contains the source code for the stored procedure.
4. Create the debug table.

If you do not use the Stored Procedure Builder to invoke the debugger, create the debug table with the following command:

```
db2 -tf sqllib/misc/db2debug.ddl
```

Note: On DB2 Enterprise - Extended Edition systems, the default nodegroup is IBMDEFAULTGROUP for the USERSPACE1 table space, and consists of all nodes defined for the system. To improve the performance of debugging stored procedures in a DB2 Enterprise - Extended Edition configuration, you should have a single coordinator node where debugging will occur and define a nodegroup that only contains that node.

5. Start the debugger daemon on the client.

   From the stored procedure client, start the debugger daemon with the following command:

   ```
   db2dbugd -qport=portno
   ```

   where *portno* is an unused TCP/IP port number. If you do not supply a value, the debugger uses 8000 as the default port number. On Windows 32-bit operating systems, you can also click the debugger daemon shortcut located in the DB2 folder to start the debugger with the default port number.

Populating the Debug Table

The debug table contains information about the stored procedures you debug and the client/server environment that you debug in. Only DBAs or users with INSERT, UPDATE, or DELETE privilege can manipulate values directly in the base table DB2DBG.ROUTINE_DEBUG. However, unless the DBA has added further restrictions, anyone can add, update, or delete rows through the user view, DB2DBG.ROUTINE_DEBUG_USER. Therefore, the rest of this section assumes that you are populating that table through the user view.

If you use the Stored Procedure Builder to invoke debugging, you can use the debugger utility to populate and manage the debug table. Otherwise, to enable debugging support for a given stored procedure, issue the following command from the CLP:

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
   ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('authid', 'S', 'schema', 'proc_name', 'Y', 'IP_num')
```

where:

authid The user name used for debugging the stored procedure, that is, the user name used to connect to the database.

schema
The schema name for the stored procedure.

proc_name
The specific name of the stored procedure. This is the specific name that was provided on the CREATE PROCEDURE command or a system generated identifier, if no specific name has been provided.

IP_num
The IP address in the form `nnn.nnn.nnn.nnn` of the client used to debug the stored procedure.

For example, to enable debugging for the stored procedure *MySchema.myProc* by the user *USER1* with the debugging client located at the IP address 123.234.111.222, type the following command:

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
   ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('USER1', 'S', 'MySchema', 'myProc', 'Y', '123.234.111.222')
```

If you drop a stored procedure, its debug information is not automatically deleted from the debug table. Debug information for non-existent stored procedures cannot harm your database or instance. If you want to keep the debug table synchronized with the DB2 catalog, you must delete the debug information manually.

Whether you create the debug table manually or through the Stored Procedure Builder, the debug table is named DB2DBG.ROUTINE_DEBUG and has the following definition:

Table 33. DB2DBG.ROUTINE_DEBUG Table Definition

Column Name	Data Type	Attributes	Description
AUTHID	VARCHAR(128)	NOT NULL, DEFAULT USER	The application authid under which the debugging for this stored procedure is to be performed. This is the user ID that was provided on connect to the database.
TYPE	CHAR(1)	NOT NULL	Valid values: 'S' (Stored Procedure)
ROUTINE_SCHEMA	VARCHAR(128)	NOT NULL	Schema name of the stored procedure to be debugged
SPECIFICNAME	VARCHAR(18)	NOT NULL	Specific name of the stored procedure to be debugged
DEBUG_ON	CHAR(1)	NOT NULL, DEFAULT 'N'	Valid values: • **Y** - enables debugging for the stored procedure named in ROUTINE_SCHEMA.SPECIFICNAME • **N** - disables debugging for stored procedure named in ROUTINE_SCHEMA.SPECIFICNAME. This is the default.

Table 33. DB2DBG.ROUTINE_DEBUG Table Definition (continued)

Column Name	Data Type	Attributes	Description
CLIENT_IPADDR	VARCHAR(15)	NOT NULL	The IP address of the client that does the debugging of the form `nnn.nnn.nnn.nnn`
CLIENT_PORT	INTEGER	NOT NULL, DEFAULT 8000	The port of the debugging communication. The default is 8000.
DEBUG_STARTN	INTEGER	NOT NULL	Not used.
DEBUG_STOPN	INTEGER	NOT NULL	Not used.
The primary key of this table is AUTHID, TYPE, ROUTINE_SCHEMA, SPECIFICNAME.			

Invoking the Debugger

If you have successfully followed the previous steps, calling a stored procedure invokes the debugger on the client with the IP address that you specified in the debug table.

In the debugger, you can step through the source code, display variables, and set breakpoints in the source code. For detailed information on using the debugger, see the debugger documentation contained in the online help.

Java Stored Procedures and UDFs

Java stored procedures and UDFs, collectively known as *Java routines*, must be registered in the DB2 catalog. DB2 Universal Database Version 7 supports the *SQLJ Routines* core specification for registering and deploying Java routines. Use PARAMETER STYLE JAVA in your CREATE PROCEDURE and CREATE FUNCTION statements to specify compliance with SQLJ Routines.

Alternatively, DB2 supports DB2 V5 and V5.2 PARAMETER STYLE DB2GENERAL stored procedures and UDFs. For more information, see "Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs" on page 765.

To register a Java function or stored procedure, follow these steps:

1. Create the Java routine as a Java method. Compile the Java source code into a Java class file. For information on creating Java stored procedures, see "Chapter 7. Stored Procedures" on page 193. For information on creating Java UDFs, see "Creating and Using Java User-Defined Functions" on page 420.
2. Collect the class file containing the Java routine in a *jar* file. You can collect one or more class files in a single JAR file. For instructions on creating JAR files, refer to the *Application Building Guide*.
3. Install the JAR file in the DB2 instance. For instructions on how to use the CALL SQLJ.INSTALL_JAR statement from the command line, see

"Installing, Replacing, and Removing JAR Files". You can also CALL the `sqlj.install_jar` procedure in an application or from the CLP.

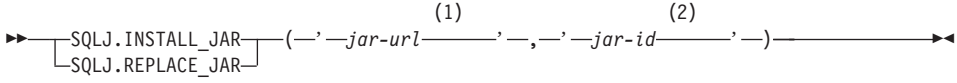
4. Issue the appropriate CREATE PROCEDURE or CREATE FUNCTION SQL statement for the Java routine.
   - For a description and examples of using the CREATE PROCEDURE statement, see "Registering Stored Procedures" on page 199.
   - For a description and examples of using the CREATE FUNCTION statement, refer to the *SQL Reference*.

When you install a JAR file, DB2 extracts the Java class files from the JAR file and registers each class in the system catalog. DB2 copies the JAR file to a `jar/schema` subdirectory of the function directory. DB2 gives the new copy of the JAR file the name given in the *jar-id* clause. Do not directly modify a JAR file which has been installed in the DB2 instance. Instead, you can use the CALL SQLJ.REMOVE_JAR and CALL SQLJ.REPLACE_JAR commands to remove or replace an installed JAR file.

Installing, Replacing, and Removing JAR Files

To install or replace a JAR file in the DB2 instance, you can use the following command syntax at the Command Line Processor:

CALL



Notes:

1 Specifies the URL containing the JAR file to be installed or replaced. The only URL scheme supported is 'file:'.

2 Specifies the JAR identifier in the database to be associated with the file specified by the jar-url.

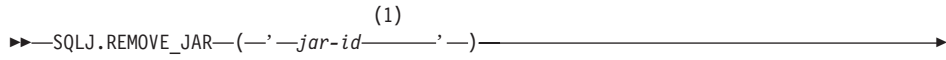
Note: On OS/2 and Windows 32-bit operating systems, DB2 stores JAR files in the path specified by the *DB2INSTPROF* instance-specific registry setting. To make JAR files unique for an instance, you must specify a unique value for *DB2INSTPROF* for that instance.

For example, to install the `Proc.jar` file located in the `file:/home/db2inst/classes/` directory in the DB2 instance, issue the following command from the Command Line Processor:

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

Subsequent SQL commands that use of the `Procedure.jar` file refer to it with the name `myproc_jar`. To remove a JAR file from the database, use the CALL REMOVE_JAR command with the following syntax:

CALL



Notes:

1 Specifies the JAR identifier of the JAR file that is to be removed from the database

To remove the JAR file `myProc_jar` from the database, enter the following command at the Command Line Processor:

```
CALL SQLJ.REMOVE_JAR('myProc_jar')
```

Function Definitions in Java

To create a Java routine, you must code the corresponding `public static` method in a `public` class. A Java routine must also be declared with the `throws SQLException` clause. Code the method signature and the rest of the method declaration to correspond with the output expected from the method body.

Functions That Return No Values in Java: To create a method that returns no values to the calling program, declare the method to return `void` and include any parameters in the signature that need to be passed to the method body. You can write a stored procedure that performs a simple UDPATE and returns no value to the client application as follows:

```
public class JavaExamples {
   public static void updateJob(String oldJob, String newJob)
         throws SQLException {
      Connection conn=DriverManager.getConnection("jdbc:ibm.db2.sample");
      PreparedStatement stmt = conn.prepareStatement("UPDATE employee
         SET job = ? WHERE job = ?");
      stmt.setString(1, newJob);
      stmt.setString(2, oldJob);
      stmt.executeUpdate();
      conn.close();
      return;
      }
   }
```

Functions That Return A Single Value in Java: Declare Java methods that return a single value with the Java return type that corresponds to the respective SQL data type (see "Supported SQL Data Types in Java" on page 639). You can write a scalar UDF that returns an SQL `INTEGER` value as follows:

```
public class JavaExamples {
   public static int getDivision(String division) throws SQLException {
      if (division.equals("Corporate")) return 1;
      else if (division.equals("Eastern")) return 2;
      else if (division.equals("Midwest")) return 3;
      else if (division.equals("Western")) return 4;
      else return 5;
   }
}
```

Functions That Return Multiple Values in Java: Java methods which are cataloged as stored procedures may return one or more values. You can also write Java stored procedures that return multiple result sets; see "Returning Result Sets from Stored Procedures" on page 233. To code a method which will return a predetermined number of values, declare the return type `void` and include the types of the expected output as arrays in the method signature. You can write a stored procedure which returns the names, years of service, and salaries of the two most senior employees with a salary under a given threshold as follows:

```
public Class JavaExamples {
   public static void lowSenioritySalary
      (String[] name1, int[] years1, BigDecimal[] salary1,
       String[] name2, int[] years2, BigDecimal[] salary2,
       Integer threshhold) throws SQLException {
      #sql iterator ByNames (String name, int years, BigDecimal salary);
      ByNames result;
      #sql result = {"SELECT name, years, salary
                      FROM staff
                      WHERE salary < :threshhold
                      ORDER BY years DESC"};
      if (result.next()) {
         name1[0] = result.name();
         years1[0] = result.years();
         salary1[0] = result.salary();
      }
      else {
         name1[0] = "****";
         return;
      }
      if (result.next()) {
         name2[0] = result.name();
         years2[0] = result.years();
         salary2[0] = result.salary();
      }
      else {
         name2[0] = "****";
```

```
            return;
          }
        }
    }
```

Using LOBs and Graphical Objects With JDBC 1.2

The JDBC 2.0 specification for JDK 1.2 defines support for LOBs and graphic types. For more information on the DB2 JDBC 2.0 driver support, see "JDBC 2.0" on page 649.

Note: You cannot use the DB2 JDBC 2.0 driver support for LOB and graphic types in stored procedures or UDFs. To use LOB or graphic types in stored procedures or UDFs, you must use the JDBC 1.2 LOB support. For more information on using using DB2 JDBC 1.2 LOB support with the DB2 JDBC 2.0 driver, see "JDBC 2.0 Compatibility" on page 651.

However, the JDBC 1.2 specification does not explicitly mention large objects (LOBs) or graphic types. DB2 provides the following support for LOBs and graphic types if you use the JDBC 1.2 driver.

If you use LOBs or graphic types in your applications, 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.

Treat GRAPHIC and DBCLOB data types as the corresponding CHAR types.

DB2 clients convert data directly from the server code page to Unicode. The following JDBC APIs convert data to or from Unicode:

getString
: Converts from server code page to Unicode.

setString
: Converts from Unicode to server code page.

getUnicodeStream
: Converts from server code page to Unicode.

setUnicodeStream
: Converts from Unicode to server code page.

The following JDBC APIs involve conversion between the client code page and the server code page:

setAsciiStream
: Converts from client code page to server code page.

getAsciiStream
Converts from server code page to client code page.

JDBC and SQLJ Interoperability

The SQLJ language provides direct support for static SQL operations that are known at the time the program is written. If some or all of a particular SQL statement cannot be determined until run time, it is a dynamic operation. To perform dynamic SQL operations from an SQLJ program, use JDBC. A ConnectionContext object contains a JDBC Connection object which can be used to create JDBC Statement objects needed for dynamic SQL operations.

Every SQLJ ConnectionContext class includes a constructor that takes as an argument a JDBC Connection. This constructor is used to create an SQLJ connection context instance that shares its underlying database connection with that of the JDBC connection.

Every SQLJ ConnectionContext instance has a `getConnection()` method that returns a JDBC Connection instance. The JDBC Connection returned shares the underlying database connection with the SQLJ connection context. It may be used to perform dynamic SQL operations as described in the JDBC API.

Session Sharing

The interoperability methods described above provide a conversion between the connection abstractions used in SQLJ and those used in JDBC. Both abstractions share the same database session, that is, the underlying database connection. Accordingly, calls to methods that affect session state on one object will also be reflected in the other object, as it is actually the underlying shared session that is being affected.

JDBC defines the default values for session state of newly created connections. In most cases, SQLJ adopts these default values. However, whereas a newly created JDBC connection has auto commit mode on by default, an SQLJ connection context requires the auto commit mode to be specified explicitly upon construction.

Connection Resource Management in Java

Calling the `close()` method of a connection context instance causes the associated JDBC connection instance and the underlying database connection to be closed. Since connection contexts may share the underlying database connection with other connection contexts and/or JDBC connections, it may not be desirable to close the underlying database connection when a connection context is closed. A programmer may wish to release the resources maintained by the connection context (for example, statement handles) without actually closing the underlying database connection. To this end, connection context classes also support a `close()` method which takes a

Boolean argument indicating whether or not to close the underlying database connection: the constant `CLOSE_CONNECTION` if the database connection should be closed, and `KEEP_CONNECTION` if it should be retained. The variant of `close()` that takes no arguments is a shorthand for calling `close(CLOSE_CONNECTION)`.

If a connection context instance is not explicitly closed before it is garbage collected, then `close(KEEP_CONNECTION)` is called by the finalize method of the connection context. This allows connection related resources to be reclaimed by the normal garbage collection process while maintaining the underlying database connection for other JDBC and SQLJ objects that may be using it. Note that if no other JDBC or SQLJ objects are using the connection, then the database connection is closed and reclaimed by the garbage collection process.

Both SQLJ connection context objects and JDBC connection objects respond to the `close()` method. When writing an SQLJ program, it is sufficient to call the `close()` method on only the connection context object. This is because closing the connection context also closes the JDBC connection associated with it. However, it is not sufficient to close only the JDBC connection returned by the `getConnection()` method of a connection context. This is because the `close()` method of a JDBC connection does not cause the containing connection context to be closed, and therefore resources maintained by the connection context are not released until it is garbage collected.

The `isClosed()` method of a connection context returns `true` if any variant of the `close()` method has been called on the connection context instance. If `isClosed()` returns `true`, then calling `close()` has no effect, and calling any other method is undefined.

Chapter 22. Programming in Perl

Programming Considerations for Perl

Perl is a popular programming language that is freely available for many operating systems. Using the DBD::DB2 driver available from http://www.ibm.com/software/data/db2/perl with the Perl Database Interface (DBI) Module available from http://www.perl.com, you can create DB2 applications using Perl.

Because Perl is an interpreted language and the Perl DBI Module uses dynamic SQL, Perl is an ideal language for quickly creating and revising prototypes of DB2 applications. The Perl DBI Module uses an interface that is quite similar to the CLI and JDBC interfaces, which makes it easy for you to port your Perl prototypes to CLI and JDBC.

Most database vendors provide a database driver for the Perl DBI Module, which means that you can also use Perl to create applications that access data from many different database servers. For example, you can write a Perl DB2 application that connects to an Oracle database using the DBD::Oracle database driver, fetch data from the Oracle database, and insert the data into a DB2 database using the DBD::DB2 database driver.

Perl Restrictions

The Perl DBI module supports only dynamic SQL. When you need to execute a statement multiple times, you can improve the performance of your Perl DB2 applications by issuing a `prepare` call to prepare the statement.

For current information on the restrictions of the version of the DBD::DB2 driver that you install on your workstation, refer to the CAVEATS file in the DBD::DB2 driver package.

Connecting to a Database Using Perl

To enable Perl to load the DBI module, you must include the following line in your DB2 application:

```
use DBI;
```

The DBI module automatically loads the DBD::DB2 driver when you create a *database handle* using the `DBI->connect` statement with the following syntax:

```
my $dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password);
```

where:

$dbhandle
represents the database handle returned by the connect statement

dbalias
represents a DB2 alias cataloged in your DB2 database directory

$userID
represents the user ID used to connect to the database

$password
represents the password for the user ID used to connect to the database

Fetching Results in Perl

Because the Perl DBI Module only supports dynamic SQL, you do not use host variables in your Perl DB2 applications. To return results from an SQL query, perform the following steps:

Step 1. Create a database handle, as described in "Connecting to a Database Using Perl" on page 675.

Step 2. Create a statement handle from the database handle. For example, you can call `prepare` with an SQL statement as a string argument to return statement handle *$sth* from the database handle, as demonstrated in the following Perl statement:

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee '
   );
```

Step 3. Execute the SQL statement by calling `execute` on the statement handle. A successful call to `execute` associates a result set with the statement handle. For example, you can execute the statement prepared in the previous example using the following Perl statement:

```
#Note: $rc represents the return code for the execute call
my $rc = $sth->execute();
```

Step 4. Fetch a row from the result set associated with the statement handle with a call to `fetchrow()`. The Perl DBI returns a row as an array with one value per column. For example, you can return all of the rows from the statement handle in the previous example using the following Perl statement:

```
while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme $lastname\n";
}
```

Parameter Markers in Perl

To enable you to execute a prepared statement using different input values for specified fields, the Perl DBI module enables you to prepare and execute a statement using parameter markers. To include a parameter marker in an SQL statement, use the question mark (?) character.

The following Perl code creates a statement handle that accepts a parameter marker for the WHERE clause of a SELECT statement. The code then executes the statement twice using the input values 25000 and 35000 to replace the parameter marker.

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee
      WHERE salary > ?'
   );

my $rc = $sth->execute(25000);

⋮

my $rc = $sth->execute(35000);
```

SQLSTATE and SQLCODE Variables in Perl

To return the SQLSTATE associated with a Perl DBI database handle or statement handle, call the `state` method. For example, to return the SQLSTATE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlstate = $dbhandle->state;
```

To return the SQLCODE associated with a Perl DBI database handle or statement handle, call the `err` method. To return the message for an SQLCODE associated with a Perl DBI database handle or statement handle, call the `errstr` method. For example, to return the SQLCODE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlcode = $dbhandle->err;
```

Perl DB2 Application Example

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]\n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname\n";
}

# check for problems which may have terminated the fetch early
warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;
```

Chapter 23. Programming in COBOL

Programming Considerations for COBOL

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. See the Micro Focus COBOL documentation for information about embedding SQL statements, language restrictions, and supported data types for host variables.

Language Restrictions in COBOL

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.

When using Micro Focus 16-bit COBOL on OS/2, precede each database manager API call with two underscores (__). The compiler then uses fully segmented addresses when passing *by reference* parameters.

Input and Output Files for COBOL

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 for COBOL

The host-language-specific include files for COBOL have the file extension `.cbl`. If you use the ″System/390 host data type support″ feature of IBM COBOL compiler, the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_i
```

If you build the DB2 sample programs with the supplied script files, you must change the include file path specified in the script files to the `cobol_i` directory and not the `cobol_a` directory.

If you do **not** use the ″System/390 host data type support″ feature of the IBM COBOL compiler, or you use an earlier version of this compiler, then the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_a
```

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 in COBOL

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 quotation marks, 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 quotation marks, as above, no extension is added to the name.

    If the file name in quotation marks 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 quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.

– When they occur inside quotation marks, 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 in COBOL

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 in COBOL

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`, `sql`, `DB2`, or `db2`, which are 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. However, if you use FILLER more than once within an SQL DECLARE section, the precompiler fails. 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

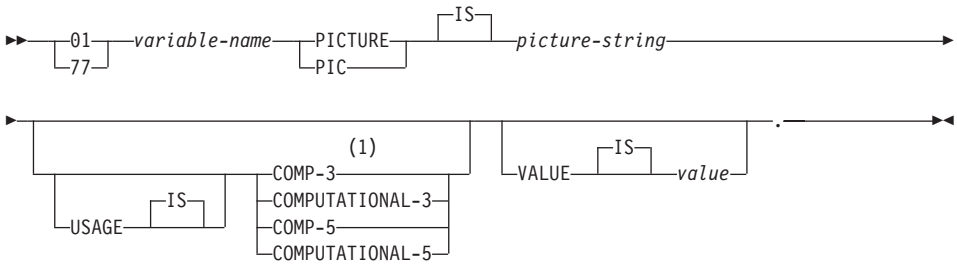
An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

For information on declaring host variables for structured types, see "Declaring Structured Type Host Variables" on page 348.

The COBOL precompiler only recognizes a subset of valid COBOL declarations.

Syntax for Numeric Host Variables in COBOL shows the syntax for numeric host variables.

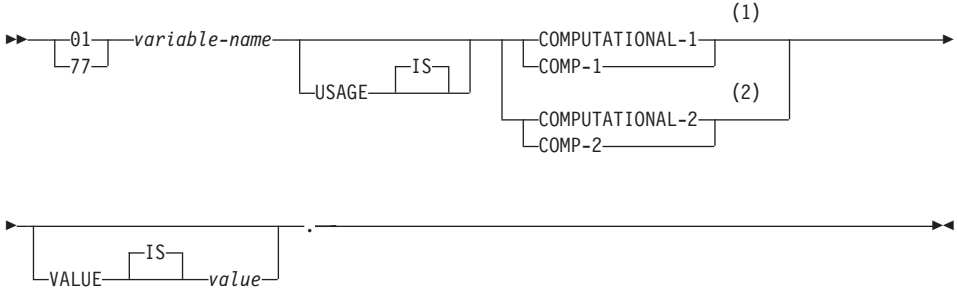
Syntax for Numeric Host Variables in COBOL



Notes:

1 An alternative for COMP-3 is PACKED-DECIMAL.

Floating Point



Notes:

1 REAL (SQLTYPE 480), Length 4

2 DOUBLE (SQLTYPE 480), Length 8

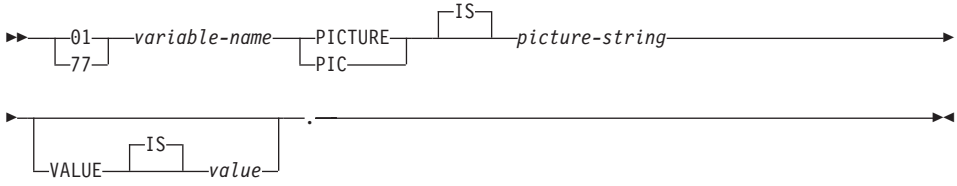
Numeric Host Variable Considerations:

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)″)

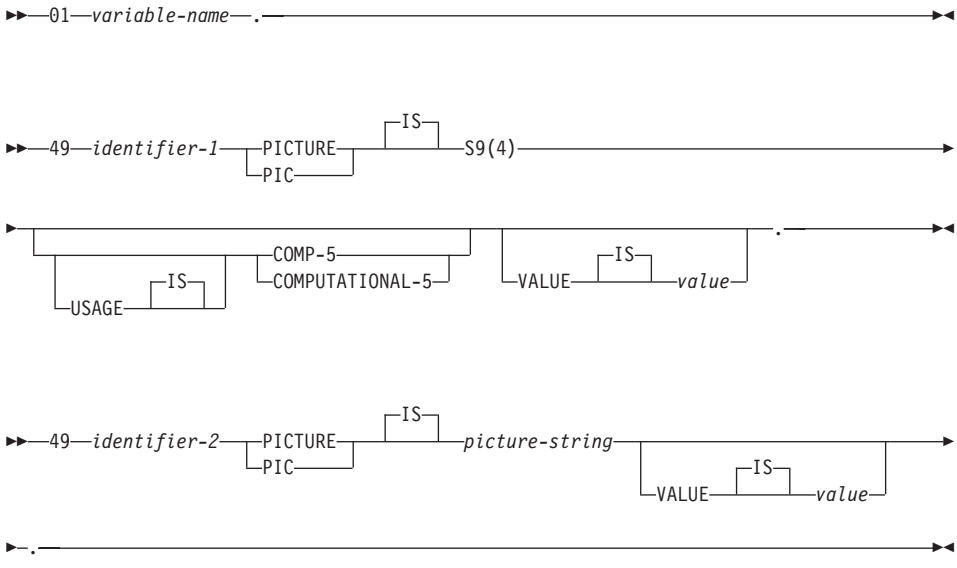
3. *m* and *n* must be positive integers.

Syntax for Character Host Variables in COBOL: Fixed Length shows the syntax for character host variables.

Syntax for Character Host Variables in COBOL: Fixed Length



Variable Length



Character Host Variable Consideration:

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 32 672, 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:

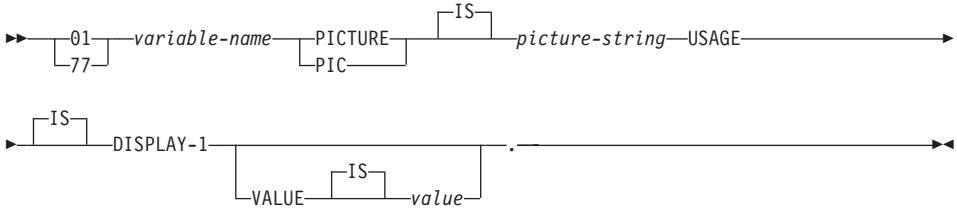
```
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

   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:

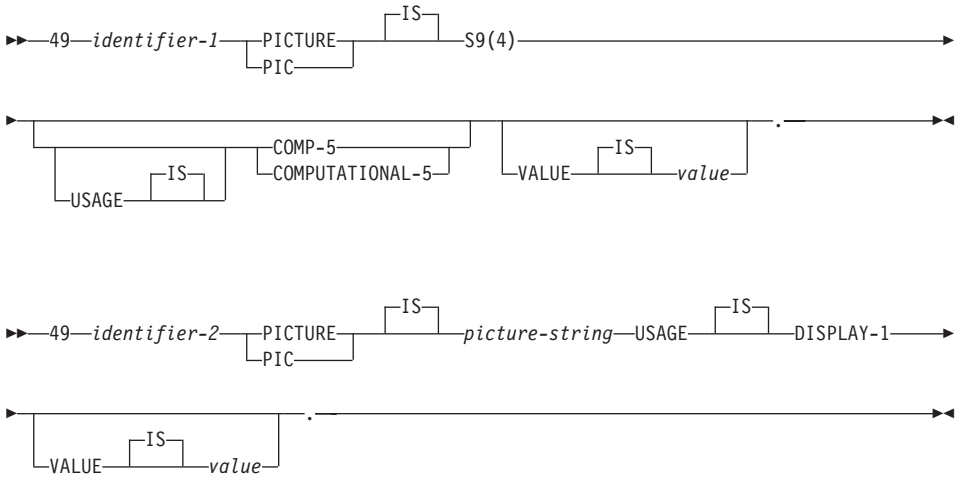
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dbname PIC X(8).
01 userid PIC X(8).
01 p-word.
    49 L PIC S9(4) COMP-5.
    49 D PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "sample" TO dbname.
    MOVE "userid" TO userid.
    MOVE "password" TO D OF p-word.
    MOVE 8          TO L of p-word.
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

Syntax for Graphic Host Variables in COBOL: Fixed Length shows the syntax for graphic host variables.

Syntax for Graphic Host Variables in COBOL: Fixed Length



Variable Length



Graphic Host Variable Considerations:

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 16 336, the host variable will be treated as a LONG VARGRAPHIC string, and its use may be restricted.

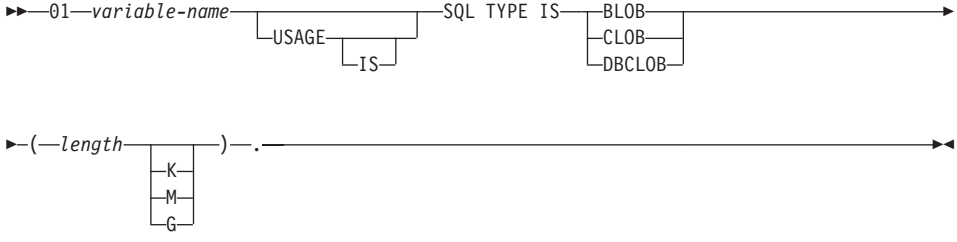
Indicator Variables in COBOL

Indicator variables should be declared as a `PIC S9(4) COMP-5` data type.

LOB Declarations in COBOL

Syntax for LOB Host Variables in COBOL shows the syntax for declaring large object (LOB) host variables in COBOL.

Syntax for LOB Host Variables in COBOL



LOB Host Variable Considerations:

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).
```

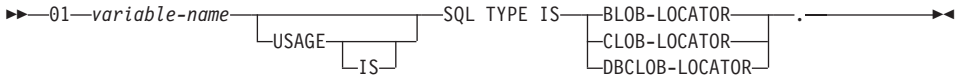
Results in the generation of the following structure:

```
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

Syntax for LOB Locator Host Variables in COBOL shows the syntax for declaring large object (LOB) locator host variables in COBOL.

Syntax for LOB Locator Host Variables in COBOL



LOB Locator Host Variable Considerations:

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.
```

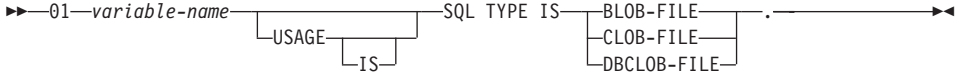
Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

File Reference Declarations in COBOL

Syntax for File Reference Host Variables in COBOL shows the syntax for declaring file reference host variables in COBOL.

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 some of the columns in the STAFF table of 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-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

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. You can nest group data items up to 10 levels. 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. In the example above, `staff-info` is a group data item, whereas `staff-name` is a VARCHAR. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. You may declare group data items at any level between 02 and 49.

You can use group data items and their subordinates in four 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
  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 names of all levels 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
  INTO
  :staff-record.staff-id,
  :staff-record.staff-name,
```

```
:staff-record.staff-info.staff-dept,
:staff-record.staff-info.staff-job
FROM staff WHERE id = 10 END-EXEC.
```

Note: The reference to `staff-id` is qualified with its group name using the prefix `staff-record.`, 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
  INTO
  :staff-id,
  :staff-name,
  :staff-dept,
  :staff-job
  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 can be uniquely identified. If, for example, `staff-job` occurs in more than one group, the precompiler issues an error indicating an ambiguous reference:

```
SQL0088N Host variable "staff-job" is ambiguous.
```

Method 4.

To resolve the ambiguous reference, you can use partial qualification of the subordinate item, for example:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-info.staff-dept,
  :staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

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, 3 and 4 above.

Indicator Tables in COBOL

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
  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=6), 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 can refer to individual elements in an indicator table in an SQL statement.**

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 ...
```

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)

Supported SQL Data Types in COBOL

Certain predefined COBOL data types correspond to column types. Only these COBOL data types can be declared as host variables.

Table 34 on page 696 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.

Note: There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 34. 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
BIGINT (492 or 493)	01 name PIC S9(18) COMP-5.	64-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*<=32 672	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*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] (920 or 921)	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 34. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
BLOB file reference variable[4] (916 or 917)	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*<=16 336	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. 16 337<=*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] (924 or 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Note:

1. The first number under **SQL 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.

- Arrays are **not** supported by the COBOL precompiler:

FOR BIT DATA in COBOL

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 in COBOL

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, and UCS-2 Considerations for COBOL

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to a the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is 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. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must

use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

For further information on these functions, refer to the *SQL Reference*. For general EUC application development guidelines, see "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 521.

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 24. Programming in FORTRAN

Programming Considerations for FORTRAN

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.

Note: FORTRAN support stabilized in DB2 Version 5, and no enhancements for FORTRAN support are planned for the future. For example, the FORTRAN precompiler cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than FORTRAN.

Language Restrictions in FORTRAN

The following sections describe the FORTRAN language restrictions.

Call by Reference in FORTRAN

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. Refer to the *Administrative API Reference* for a full description of these APIs.

Debugging and Comment Lines in FORTRAN

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.

Precompiling Considerations for FORTRAN

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.

Refer to the *Application Building Guide* for information on any other precompiling considerations that may affect you.

Input and Output Files for FORTRAN

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 for FORTRAN

The host-language specific include files for FORTRAN have the file extension `.f` on UNIX platforms, and `.for` on OS/2. You can use the following FORTRAN include files 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 147 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.

Including Files in FORTRAN

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 quotation marks, 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 quotation marks, 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 quotation marks 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.

Embedding SQL Statements in FORTRAN

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 quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
– When they occur inside quotation marks, 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 in FORTRAN

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 in FORTRAN

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 `SQL`, `sql`, `DB2`, or `db2`, which are reserved for system use.

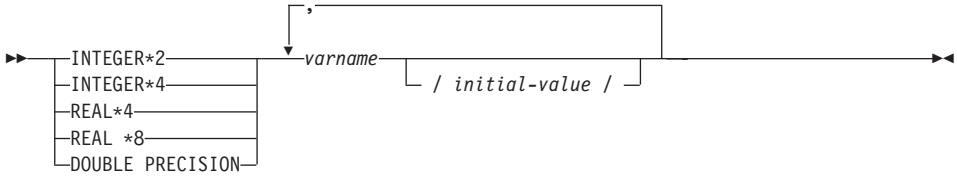
Declaring Host Variables

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. Syntax for Numeric Host Variables in FORTRAN shows the syntax for numeric host variables.

For information on declaring host variables for structured types, see "Declaring Structured Type Host Variables" on page 348.

Syntax for Numeric Host Variables in FORTRAN

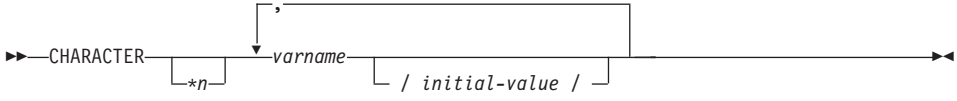


Numeric Host Variable Considerations:

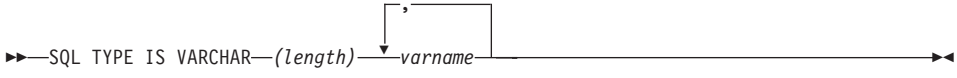
1. REAL*8 and DOUBLE PRECISION are equivalent.
2. Use an E rather than a D as the exponent indicator for REAL*8 constants.

Syntax for Character Host Variables in FORTRAN: Fixed Length shows the syntax for character host variables.

Syntax for Character Host Variables in FORTRAN: Fixed Length



Variable Length



Character Host Variable Considerations:

1. *n has a maximum value of 254.
2. When length is between 1 and 32 672 inclusive, the host variable has type VARCHAR(SQLTYPE 448).
3. When length is between 32 673 and 32 700 inclusive, the host variable has type LONG VARCHAR(SQLTYPE 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
```

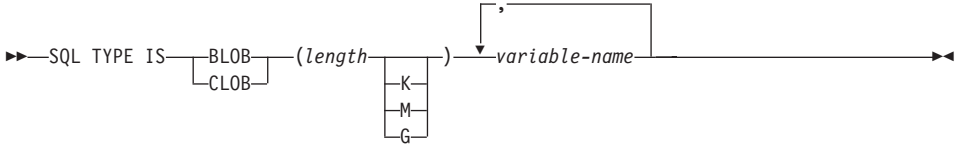
Indicator Variables in FORTRAN

Indicator variables should be declared as an INTEGER*2 data type.

LOB Declarations in FORTRAN

Syntax for Large Object (LOB) Host Variables in FORTRAN shows the syntax for declaring large object (LOB) host variables in FORTRAN.

Syntax for Large Object (LOB) Host Variables in FORTRAN



LOB Host Variable Considerations:

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
```

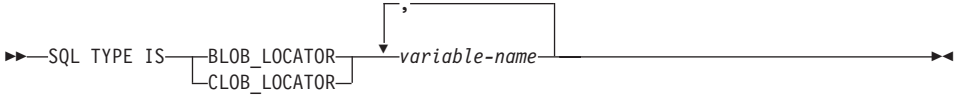
Results in the generation of the following structure:

```
 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

Syntax for Large Object (LOB) Locator Host Variables in FORTRAN shows the syntax for declaring large object (LOB) locator host variables in FORTRAN.

Syntax for Large Object (LOB) Locator Host Variables in FORTRAN



LOB Locator Host Variable Considerations:

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
```

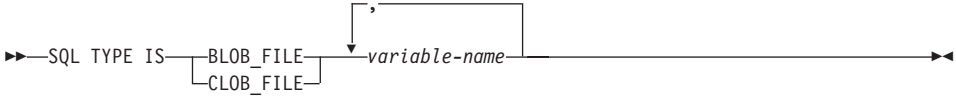
Results in the generation of the following declaration:

```
integer*4 my_locator
```

File Reference Declarations in FORTRAN

Syntax for File Reference Host Variables in FORTRAN shows the syntax for declaring file reference host variables in FORTRAN.

Syntax for File Reference Host Variables in FORTRAN



File Reference Host Variable Considerations:

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 in FORTRAN

Certain predefined FORTRAN data types correspond to database manager column types. Only these FORTRAN data types can be declared as host variables.

Table 35 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.

Note: There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 35. 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
INTEGER (496 or 497)	INTEGER*4	32-bit, signed integer
REAL[2] (480 or 481)	REAL*4	Single precision floating point

Table 35. SQL Data Types Mapped to FORTRAN Declarations (continued)

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
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 32 672	Variable-length character string
LONG VARCHAR (456 or 457)	SQL TYPE IS VARCHAR(*n*) where *n* is from 32 673 to 32 700	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] (920 or 921)	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] (916 or 917)	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

Note:

1. The first number under **SQL 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 in FORTRAN

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 in FORTRAN

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, and UCS-2 Considerations for FORTRAN

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to a the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is 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. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

For further information on these functions, refer to the *SQL Reference*.

For general EUC application development guidelines, see "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 521.

Chapter 25. Programming in REXX

Programming Considerations for REXX

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.

Note: REXX support stabilized in DB2 Version 5, and no enhancements for REXX support are planned for the future. For example, REXX cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than REXX.

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 for REXX" on page 730 for details and restrictions on how this interface can be used.

Language Restrictions for REXX

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 quotation marks (' ' or " "). If you do not use quotation marks, 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 and Windows 32-bit operating systems, but not on AIX.

Registering SQLEXEC, SQLDBS and SQLDB2 in REXX

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 in REXX

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 quotation marks, 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 in REXX

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 in REXX

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 in REXX

The REXX interpreter examines every string without quotation marks 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 quotation marks as in the following example:

```
VAR = '100'
```

REXX sets the variable *VAR* to the 3-byte character string `100`. If single quotation marks 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 quotation marks.

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 host or AS/400 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 *Administrative 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 *Administrative 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 *Administrative 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 *Administrative 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.

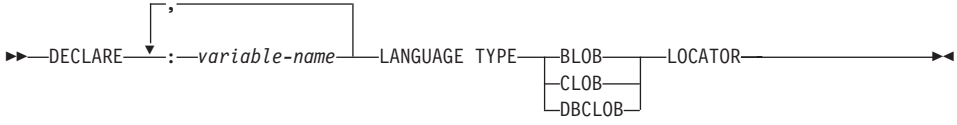
In REXX SQL, LOB types are determined from the string content of your host variable as follows:

Host variable string content	Resulting LOB type
:hv1='ordinary quoted string longer than 32K ...'	CLOB
:hv2="'string with embedded delimiting quotation marks ", "longer than 32K...'"	CLOB
:hv3="G'DBCS string with embedded delimiting single ", "quotation marks, beginning with G, longer than 32K...'"	DBCLOB
:hv4="BIN'string with embedded delimiting single ", "quotation marks, beginning with BIN, any length...'"	BLOB

LOB Locator Declarations in REXX

"Syntax for LOB Locator Host Variables in REXX" shows the syntax for declaring LOB locator host variables in REXX.

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:

Syntax for FREE LOCATOR Statement

Example:

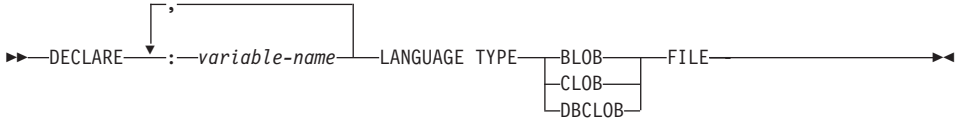
```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

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.

"Syntax for LOB File Reference Variables in REXX" shows the syntax for declaring LOB file reference host variables in REXX.

REXX File Reference Declarations



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 in REXX

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 in REXX

Certain predefined REXX data types correspond to DB2 column types. Table 36 shows how SQLEXEC and SQLDBS interpret REXX variables in order to convert their contents to DB2 data types.

Note: There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 36. 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
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] (920 or 921)	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] (916 or 917)	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

Table 36. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type[1]	REXX Data Type	SQL Column Type Description
TIME (388 or 389)	Equivalent to CHAR(*8*)	8-byte character string
TIMESTAMP (392 or 393)	Equivalent to CHAR(*26*)	26-byte character string
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] (924 or 925)	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 Windows 32-bit operating systems, 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_07_01/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 host or AS/400 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 for REXX

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 *Administrative API Reference*.

If a DB2 API you want to use cannot be called using the SQLDBS routine (and consequently, not listed in the *Administrative 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). Refer to 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(1) (2)
Bind for DB2 Version 2 or 5(1)
Create Database at Node
Drop Database at Node
Drop Node Verify
Deactivate Database
Deregister
Load(3)
Load Query
Precompile Program(1)
Rebind Package(1)
Redistribute Nodegroup
Register
Start Database Manager
Stop Database Manager

Notes on DB2 APIs Supported by SQLDB2:

1. 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.
2. Is supported on OS/2 through the SQLDB2 interface.
3. The optional output parameter, `pLoadInfoOut` for the Load API is not returned to the application in REXX. Refer to the *Administrative 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 in REXX

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 37 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 37.

Table 37. 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 for REXX

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.

Considerations on the Server for REXX

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:

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
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 for REXX

REXX applications are not supported under Japanese or Traditional Chinese EUC environments.

Part 7. Appendixes

Appendix A. Supported SQL Statements

Table 38:
- Lists all the supported SQL statements in DB2 Universal Database for Linux, OS/2, UNIX, and Windows 32-bit operating systems
- 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 DB2 CLI function name) if the statement can be executed using the DB2 Call Level Interface (DB2 CLI)
- Indicates (with an 'X') if the statement can be executed in an SQL procedure

You can use Table 38 as a quick reference aid. For a complete discussion of all the statements, including their syntax, refer to the *SQL Reference*.

Table 38. SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
ALLOCATE CURSOR				X
assignment statement				X
ASSOCIATE LOCATORS				X
ALTER { BUFFERPOOL, NICKNAME,[10] NODEGROUP, SERVER,[10] TABLE, TABLESPACE, USER MAPPING,[10] TYPE, VIEW }	X	X	X	
BEGIN DECLARE SECTION[2]				
CALL		X[9]	X[4]	X
CASE statement				X
CLOSE		X	`SQLCloseCursor()`, `SQLFreeStmt()`	X
COMMENT ON	X	X	X	X
COMMIT	X	X	`SQLEndTran`, `SQLTransact()`	X
Compound SQL (Embedded)			X[4]	
compound statement				X

Table 38. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
CONNECT (Type 1)		X	`SQLBrowseConnect()`, `SQLConnect()`, `SQLDriverConnect()`	
CONNECT (Type 2)		X	`SQLBrowseConnect()`, `SQLConnect()`, `SQLDriverConnect()`	
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING,[10] INDEX, INDEX EXTENSION, METHOD, NICKNAME,[10] NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRANSFORM TYPE MAPPING,[1] TRIGGER, USER MAPPING,[10] TYPE, VIEW, WRAPPER[10] }	X	X	X	X[11]
DECLARE CURSOR[2]		X	`SQLAllocStmt()`	X
DECLARE GLOBAL TEMPORARY TABLE	X	X	X	X
DELETE	X	X	X	X
DESCRIBE[8]		X	`SQLColAttributes()`, `SQLDescribeCol()`, `SQLDescribParam()`[6]	
DISCONNECT		X	`SQLDisconnect()`	
DROP	X	X	X	X[11]
END DECLARE SECTION[2]				
EXECUTE			`SQLExecute()`	X
EXECUTE IMMEDIATE			`SQLExecDirect()`	X
EXPLAIN	X	X	X	X
FETCH		X	`SQLExtendedFetch()`[7], `SQLFetch()`, `SQLFetchScroll()`[7]	X
FLUSH EVENT MONITOR	X	X	X	
FOR statement				X

Table 38. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
FREE LOCATOR			X[4]	X
GET DIAGNOSTICS				X
GOTO statement				X
GRANT	X	X	X	X
IF statement				X
INCLUDE[2]				
INSERT	X	X	X	X
ITERATE				X
LEAVE statement				X
LOCK TABLE	X	X	X	X
LOOP statement				X
OPEN		X	`SQLExecute()`, `SQLExecDirect()`	X
PREPARE			`SQLPrepare()`	X
REFRESH TABLE	X	X	X	
RELEASE		X		X
RELEASE SAVEPOINT	X	X	X	X
RENAME TABLE	X	X	X	
RENAME TABLESPACE	X	X	X	
REPEAT statement				X
RESIGNAL statement				X
RETURN statement				X
REVOKE	X	X	X	
ROLLBACK	X	X	`SQLEndTran()`, `SQLTransact()`	X
SAVEPOINT	X	X	X	X
select-statement	X	X	X	X
SELECT INTO				X
SET CONNECTION		X	`SQLSetConnection()`	
SET CURRENT DEFAULT TRANSFORM GROUP	X	X	X	X
SET CURRENT DEGREE	X	X	X	X

Table 38. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
SET CURRENT EXPLAIN MODE	X	X	X, `SQLSetConnectAttr()`	X
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, `SQLSetConnectAttr()`	X
SET CURRENT PACKAGESET				
SET CURRENT QUERY OPTIMIZATION	X	X	X	X
SET CURRENT REFRESH AGE	X	X	X	X
SET EVENT MONITOR STATE	X	X	X	X
SET INTEGRITY	X	X	X	
SET PASSTHRU[10]	X	X	X	X
SET PATH	X	X	X	X
SET SCHEMA	X	X	X	X
SET SERVER OPTION[10]	X	X	X	X
SET transition-variable[5]	X	X	X	X
SIGNAL statement				X
SIGNAL SQLSTATE[5]	X	X	X	
UPDATE	X	X	X	X
VALUES INTO				X
WHENEVER[2]				
WHILE statement				X

Table 38. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure

Note:

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*.
9. When CALL is issued through the command line processor, only certain procedures and their respective parameters are supported (see page). (see "Installing, Replacing, and Removing JAR Files" on page 669). .
10. Statement is supported only for federated database servers.
11. SQL procedures can only issue CREATE and DROP statements for indexes, tables, and views.

Appendix B. Sample Programs

This section provides information on the sample programs supplied with DB2. All sample programs can be found in the `samples` subdirectory of the `sqllib` directory. There is a subdirectory for each supported language.

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 116. See the *Application Building Guide* for details on how to install, build, and execute these programs in your environment.

Notes:

1. This section describes sample programs for the programming languages for all platforms supported by DB2. Not all sample programs have been ported to all platforms or supported programming languages.
2. DB2 sample programs are provided "as is" without any warranty of any kind. The user, and not IBM, assumes the entire risk of quality, performance, and repair of any defects.

The sample programs come with the DB2 Application Development (DB2 AD) Client. You can use the sample programs as templates to create your own applications.

Sample program file extensions differ for each supported language, and for embedded SQL and non-embedded SQL programs within each language. File extensions may also differ for groups of programs within a language. These different sample file extensions are categorized in the following tables:

Sample File Extensions by Language
Table 39 on page 745.

Sample File Extensions by Program Group
Table 40 on page 745.

The following tables document the sample programs by type:

DB2 API Sample Programs with No Embedded SQL
Table 41 on page 747.

DB2 API Embedded SQL Sample Programs
Table 42 on page 750.

Embedded SQL Sample Programs with No DB2 APIs
Table 43 on page 752.

User-Defined Function Sample Programs
Table 44 on page 754

DB2 CLI Sample Programs
Table 45 on page 754.

Java JDBC Sample Programs
Table 46 on page 756.

Java SQLJ Sample Programs
Table 47 on page 757.

SQL Procedure Sample Programs
Table 48 on page 758.

ActiveX Data Objects, Remote Data Objects, and Microsoft Transaction Server Sample Programs
Table 49 on page 760.

Object Linking and Embedding (OLE) Automation Sample Programs
Table 50 on page 761.

Object Linking and Embedding Database (OLE DB) Table Functions
Table 51 on page 761.

Command Line Processor (CLP) Sample Programs
Table 52 on page 762.

Log Management User Exit Programs
Table 53 on page 763.

Notes:

1. Table 42 on page 750 contains programs that have both DB2 APIs and embedded SQL statements. For all DB2 API sample programs, please see both Table 41 on page 747 and Table 42 on page 750. For all embedded SQL sample programs (except for Java SQLJ), please see both Table 42 on page 750 and Table 43 on page 752.
2. Table 44 on page 754 of UDF sample programs does not contain DB2 CLI UDF programs. For these, please see Table 45 on page 754.

Table 39. Sample File Extensions by Language

Language	Directory	Embedded SQL Programs	Non-embedded SQL Programs
C	samples/c samples/cli (CLI programs)	.sqc	.c
C++	samples/cpp	.sqC (UNIX) .sqx (Windows & OS/2)	.C (UNIX) .cxx (Windows & OS/2)
COBOL	samples/cobol samples/cobol_mf	.sqb	.cbl
JAVA	samples/java	.sqlj	.java
REXX	samples/rexx	.cmd	.cmd

Table 40. Sample File Extensions by Program Group

Sample Group	Directory	File Extension
ADO, RDO, MTS	samples\ADO\VB (Visual Basic) samples\ADO\VC (Visual C++) samples\RDO samples\MTS	.bas .frm .vbp (Visual Basic) .cpp .dsp .dsw (Visual C++)
CLP	samples/clp	.db2
OLE	samples\ole\msvb (Visual Basic) samples\ole\msvc (Visual C++)	.bas .vbp (Visual Basic) .cpp (Visual C++)
OLE DB	samples\oledb	.db2
SQL Procedures	samples/sqlproc	.db2 .c .sqc (Client Applications)
User Exit	samples/c	.cad (OS/2) .cadsm (UNIX & Windows) .cdisk (UNIX & Windows) .ctape (UNIX)

Note:

Directory Delimiters
On UNIX are /. On OS/2 and Windows platforms, are \. In the tables, the UNIX delimiters are used unless the directory is only available on Windows and/or OS/2.

File Extensions
Are provided for the samples in the tables where only one extension exists.

Embedded SQL Programs
Require precompilation, except for REXX embedded SQL programs where the embedded SQL statements are interpreted when the program is run.

IBM COBOL samples
Are only supplied for AIX, OS/2, and Windows 32-bit operating systems in the `cobol` subdirectory.

Micro Focus Cobol Samples
Are only supplied for AIX, HP-UX, OS/2, Solaris Operating Environment, and Windows 32-bit operating systems in the `cobol_mf` subdirectory.

Java Samples
Are Java Database Connectivity (JDBC) applets, applications, and stored procedures, embedded SQL for Java (SQLJ) applets, applications, and stored procedures, as well as Java UDFs. Java samples are available on all supported DB2 platforms.

REXX Samples
Are only supplied for AIX, OS/2, and Windows NT operating systems.

CLP Samples
Are Command Line Processor scripts that execute SQL statements.

OLE Samples
Are for Object Linking and Embedding (OLE) in Microsoft Visual Basic and Microsoft Visual C++, supplied for Windows 32-bit operating systems only.

ADO, RDO, and MTS Samples
Are ActiveX Data Objects samples in Microsoft Visual Basic and Microsoft Visual C++, and Remote Data Objects and Microsoft Transaction Server samples in Microsoft Visual Basic, supplied for Windows 32-bit operating systems only.

User Exit samples
Are Log Management User Exit programs used to archive and retrieve database log files. The files must be renamed with a `.c` extension and compiled as C language programs.

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 programs in the samples tables, refer to the `README` file in the appropriate `samples` subdirectory under your DB2 instance. The `README` file will contain any additional samples that are not listed in this book.

- **On OS/2 and Windows 32-bit operating systems.**

  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 the drive where DB2 is installed, `%DB2PATH%` will point to *drive*:`\sqllib`. For additional information about the sample programs in the samples tables, 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.

If your platform is not addressed in Table 39 on page 745, please refer to the *Application Building Guide* for information specific to your environment.

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.

DB2 API Non-Embedded SQL Samples

Table 41. DB2 API Sample Programs with No Embedded SQL

Sample Program	Included APIs
backrest	• sqlbftcq - Fetch Tablespace Container Query • sqlbstsc - Set Tablespace Containers • sqlfudb - Update Database Configuration • sqlubkp - Backup Database • sqluroll - Rollforward Database • sqlurst - Restore Database
checkerr	• sqlaintp - Get Error Message • sqlogstt - Get SQLSTATE Message
cli_info	• sqleqryi - Query Client Information • sqleseti - Set Client Information
client	• sqleqryc - Query Client • sqlesetc - Set Client

Table 41. DB2 API Sample Programs with No Embedded SQL (continued)

Sample Program	Included APIs
d_dbconf	• sqleatin - Attach • sqledtin - Detach • sqlfddb - Get Database Configuration Defaults
d_dbmcon	• sqleatin - Attach • sqledtin - Detach • sqlfdsys - Get Database Manager Configuration Defaults
db_udcs	• sqleatin - Attach • sqlecrea - Create Database • sqledrpd - Drop Database
db2mon	• sqleatin - Attach • sqlmon - Get/Update Monitor Switches • sqlmonss - Get Snapshot • sqlmonsz - Estimate Size Required for sqlmonss() Output Buffer • sqlmrset - Reset Monitor
dbcat	• sqlecadb - Catalog Database • sqledcls - Close Database Directory Scan • sqledgne - Get Next Database Directory Entry • sqledosd - Open Database Directory Scan • sqleuncd - Uncatalog Database
dbcmt	• sqledcgd - Change Database Comment • sqledcls - Close Database Directory Scan • sqledgne - Get Next Database Directory Entry • sqledosd - Open Database Directory Scan • sqleisig - Install Signal Handler
dbconf	• sqleatin - Attach • sqlecrea - Create Database • sqledrpd - Drop Database • sqlfrdb - Reset Database Configuration • sqlfudb - Update Database Configuration • sqlfxdb - Get Database Configuration

Table 41. DB2 API Sample Programs with No Embedded SQL (continued)

Sample Program	Included APIs
dbinst	• sqleatcp - Attach and Change Password • sqleatin - Attach • sqledtin - Detach • sqlegins - Get Instance
dbmconf	• sqleatin - Attach • sqledtin - Detach • sqlfrsys - Reset Database Manager Configuration • sqlfusys - Update Database Manager Configuration • sqlfxsys - Get Database Manager Configuration
dbsnap	• sqleatin - Attach • sqlmonss - Get Snapshot
dbstart	• sqlepstart - Start Database Manager
dbstop	• sqlefrce - Force Application • sqlepstp - Stop Database Manager
dcscat	• sqlegdad - Catalog DCS Database • sqlegdcl - Close DCS Directory Scan • sqlegdel - Uncatalog DCS Database • sqlegdge - Get DCS Directory Entry for Database • sqlegdgt - Get DCS Directory Entries • sqlegdsc - Open DCS Directory Scan
dmscont	• sqleatin - Attach • sqlecrea - Create Database • sqledrpd - Drop Database
ebcdicdb	• sqleatin - Attach • sqlecrea - Create Database • sqledrpd - Drop Database
migrate	• sqlemgdb - Migrate Database
monreset	• sqleatin - Attach • sqlmrset - Reset Monitor
monsz	• sqleatin - Attach • sqlmonss - Get Snapshot • sqlmonsz - Estimate Size Required for sqlmonss() Output Buffer

Table 41. DB2 API Sample Programs with No Embedded SQL (continued)

Sample Program	Included APIs
nodecat	• sqlectnd - Catalog Node • sqlencls - Close Node Directory Scan • sqlengne - Get Next Node Directory Entry • sqlenops - Open Node Directory Scan • sqleuncn - Uncatalog Node
restart	• sqlerstd - Restart Database
setact	• sqlesact - Set Accounting String
setrundg	• sqlesdeg - Set Runtime Degree
sws	• sqleatin - Attach • sqlmon - Get/Update Monitor Switches
utilapi	• sqlaintp - Get Error Message • sqlogstt - Get SQLSTATE Message

DB2 API Embedded SQL Samples

Table 42. DB2 API Embedded SQL Sample Programs

Sample Program	Included APIs
asynrlog	• sqlurlog - Asynchronous Read Log
autocfg	• db2AutoConfig -- Autoconfig • db2AutoConfigMemory -- Autoconfig Free Memory • sqlfudb -- Update Database Configuration • sqlfusys -- Update Database Manager Configuration • sqlesetc -- Set Client • sqlaintp -- SQLCA Message
dbauth	• sqluadau - Get Authorizations
dbstat	• sqlureot - Reorganize Table • sqlustat - Runstats
expsamp	• sqluexpr - Export • sqluimpr - Import
impexp	• sqluexpr - Export • sqluimpr - Import
loadqry	• db2LoadQuery - Load Query

Table 42. DB2 API Embedded SQL Sample Programs (continued)

Sample Program	Included APIs
makeapi	• sqlabndx - Bind • sqlaprep - Precompile Program • sqlepstp - Stop Database Manager • sqlepstr - Start Database Manager
rebind	• sqlarbnd - Rebind
rechist	• sqlubkp - Backup Database • sqluhcls - Close Recovery History File Scan • sqluhgne - Get Next Recovery History File Entry • sqluhops - Open Recovery History File Scan • sqluhprn - Prune Recovery History File • sqluhupd - Update Recovery History File
tabscont	• sqlbctcq - Close Tablespace Container Query • sqlbftcq - Fetch Tablespace Container Query • sqlbotcq - Open Tablespace Container Query • sqlbtcq - Tablespace Container Query • sqlefmem - Free Memory
tabspace	• sqlbctsq - Close Tablespace Query • sqlbftpq - Fetch Tablespace Query • sqlbgtss - Get Tablespace Statistics • sqlbmtsq - Tablespace Query • sqlbotsq - Open Tablespace Query • sqlbstpq - Single Tablespace Query • sqlefmem - Free Memory
tload	• sqluexpr - Export • sqluload - Load • sqluvqdp - Quiesce Tablespaces for Table

Table 42. DB2 API Embedded SQL Sample Programs (continued)

Sample Program	Included APIs
tspace	• sqlbctcq - Close Tablespace Container Query • sqlbctsq - Close Tablespace Query • sqlbftcq - Fetch Tablespace Container Query • sqlbftpq - Fetch Tablespace Query • sqlbgtss - Get Tablespace Statistics • sqlbmtsq - Tablespace Query • sqlbotcq - Open Tablespace Container Query • sqlbotsq - Open Tablespace Query • sqlbstpq - Single Tablespace Query • sqlbstsc - Set Tablespace Containers • sqlbtcq - Tablespace Container Query • sqlefmem - Free Memory
utilemb	• sqlaintp - Get Error Message • sqlogstt - Get SQLSTATE Message

Embedded SQL Samples With No DB2 APIs

Table 43. Embedded SQL Sample programs with No DB2 APIs

Sample Program Name	Program Description
`adhoc`	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 154 for details.
`advsql`	Demonstrates the use of advanced SQL expressions like CASE, CAST, and scalar full selects.
`blobfile`	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 this file can be displayed using an external viewer.
`columns`	Demonstrates the use of a cursor that is processed using dynamic SQL. This program lists a result set from SYSCAT.COLUMNS under a desired schema name.
`cursor`	Demonstrates the use of a cursor using static SQL. See "Example: Cursor Program" on page 84 for details.
`delet`	Demonstrates static SQL to delete items from a database.
`dynamic`	Demonstrates the use of a cursor using dynamic SQL.
`joinsql`	Demonstrates using advanced SQL join expressions.

Table 43. Embedded SQL Sample programs with No DB2 APIs (continued)

Sample Program Name	Program Description
largevol	Demonstrates parallel query processing in a partitioned environment, and the use of an NFS file system to automate the merging of the result sets. Only available on AIX. See "Example: Extracting Large Volume of Data (largevol.c)" on page 562 for details.
lobeval	Demonstrates the use of LOB locators and defers the evaluation of the actual LOB data. See "How the Sample LOBEVAL Program Works" on page 360 for details.
lobfile	Demonstrates the use of LOB file handles. See "How the Sample LOBFILE Program Works" on page 368 for details.
lobloc	Demonstrates the use of LOB locators. See "How the Sample LOBLOC Program Works" on page 353 for details.
lobval	Demonstrates the use of LOBs.
openftch	Demonstrates fetching, updating, and deleting of rows using static SQL. See "How the OPENFTCH Program Works" on page 93 for details.
recursql	Demonstrates the use of advanced SQL recursive queries.
sampudf	Demonstrates User-Defined Types (UDTs) and User-Defined Functions (UDFs) implemented to modify table entries. All UDFs declared in this program are sourced UDFs.
spclient	A client application that calls stored procedures in the spserver shared library.
spcreate.db2	A CLP script that contains the CREATE PROCEDURE statements to register the stored procedures created by the spserver program.
spdrop.db2	A CLP script that contains the DROP PROCEDURE statements necessary for deregistering the stored procedures created by the spserver program.
spserver	A server program demonstrating stored procedures. The client program is spclient.
static	Demonstrates static SQL to retrieve information. See "Example: Static SQL Program" on page 63 for details.
tabsql	Demonstrates the use of advanced SQL table expressions.
tbdefine	Demonstrates creating and dropping tables.
thdsrver	Demonstrates the use of POSIX threads APIs for thread creation and management. The program maintains a pool of contexts. A generate_work function is executed from main, and creates dynamic SQL statements that are executed by worker threads. When a context becomes available, a thread is created and dispatched to do the specified work. The work generated consists of statements to delete entries from either the STAFF or EMPLOYEE tables of the sample database. This program is only available on UNIX platforms.
trigsql	Demonstrates using advanced SQL triggers and constraints.
udfcli	Demonstrates calling a user-defined function (UDF) created by the udfsrv program, and stored on the server to access tables in the sample database.

Table 43. Embedded SQL Sample programs with No DB2 APIs (continued)

Sample Program Name	Program Description
updat	Demonstrates static SQL to update a database. See "Example: UPDAT Program" on page 105 for details.
varinp	Demonstrates variable input to Embedded Dynamic SQL statement calls using parameter markers. See "How the VARINP Program Works" on page 162 for details.

User-Defined Function Samples

Table 44. User-Defined Function Sample programs

Sample Program Name	Program Description
DB2Udf.java	A Java UDF that demonstrates several tasks, including integer division, manipulation of Character Large Objects (CLOBs), and the use of Java instance variables.
udfsrv.c	Creates a library with the User-Defined Function ScalarUDF, to access the sample database tables.
UDFsrv.java	Demonstrates the use of Java User-Defined Functions (UDFs).

DB2 Call Level Interface Samples

Table 45. Sample CLI Programs in DB2 Universal Database

Sample Program Name	Program Description
Common Utility Files	
utilcli.c	Utility functions used in CLI samples.
utilapi.c	Utility functions that call DB2 APIs.
Application Level - Samples that deal with the application level of DB2 and CLI.	
apinfo.c	How to get and set application level information.
aphndls.c	How to allocate and free handles.
apsqlca.c	How to work with SQLCA data.
Installation Image Level - Samples that deal with the installation image level of DB2 and CLI.	
ilinfo.c	How to get and set installation level information (such as the version of the CLI driver).
Instance Level - Samples that deal with the instance level of DB2 and CLI.	
ininfo.c	How to get and set instance level information.

Table 45. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description
Database Level - Samples that deal with database objects in DB2.	
`dbconn.c`	How to connect and disconnect from a database.
`dbinfo.c`	How to get and set information at a database level.
`dbmconn.c`	How to connect and disconnect from multiple databases (uses DB2 APIs to create and drop second database).
`dbmuse.c`	How to perform transactions with multiple databases (uses DB2 APIs to create and drop second database).
`dbnative.c`	How to translate a statement that contains an ODBC escape clause to a data source specific format.
`dbuse.c`	How to work with database objects.
`dbusemx.sqc`	How to use a single database in conjunction with embedded SQL.
Table Level - Samples that deal with table objects in DB2.	
`tbconstr.c`	How to work with table constraints.
`tbconstr.c`	How to create, alter and drop tables.
`tbinfo.c`	How to get and set information at a table level.
`tbmod.c`	How to modify information in a table.
`tbread.c`	How to read information in a table.
Data Type Level - Samples that deal with data types.	
`dtinfo.c`	How to get information about data types.
`dtlob.c`	How to read and write LOB data.
`dtudt.c`	How to create, use, and drop user defined distinct types.
UDF Level - Samples that demonstrate user defined functions.	
`udfcli.c`	Client application which calls the user defined function in udfsrv.c.
`udfsrv.c`	User defined function ScalarUDF called by udfcli.c sample.
Stored Procedure Level - Samples that demonstrate stored procedures in CLI.	
`spcreate.db2`	CLP script to issue CREATE PROCEDURE statements.
`spdrop.db2`	CLP script to drop stored procedures from the catalog.
`spclient.c`	Client program used to call the server functions declared in spserver.c.
`spserver.c`	Stored procedure functions built and run on the server.
`spcall.c`	Program to call any stored procedure.

Table 45. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description

Note: Other files in the `samples/cli` directory include:
- `README` - Lists all example files.
- `makefile` - Makefile for all files
- build files for applications and stored procedures

Java Samples

Table 46. Java Database Connectivity (JDBC) Sample Programs

Sample Program Name	Program Description
`DB2Appl.java`	A JDBC application that queries the sample database using the invoking user's privileges.
`DB2Applt.java`	A JDBC applet that queries the database using the JDBC applet driver. It uses the user name, password, server, and port number parameters specified in DB2Applt.html.
`DB2Applt.html`	An HTML file that embeds the applet sample program, `DB2Applt`. It needs to be customized with server and user information.
`DB2UdCli.java`	A Java client application that calls the Java user-defined function, `DB2Udf`.
`Dynamic.java`	Demonstrates a cursor using dynamic SQL.
`MRSPcli.java`	This is the client program that calls the server program `MRSPsrv`. The program demonstrates multiple result sets being returned from a Java stored procedure.
`MRSPsrv.java`	This is the server program that is called by the client program, `MRSPcli`. The program demonstrates multiple result sets being returned from a Java stored procedure.
`Outcli.java`	A Java client application that calls the SQLJ stored procedure, `Outsrv`.
`PluginEx.java`	A Java program that adds new menu items and toolbar buttons to the DB2 Web Control Center.
`Spclient.java`	A JDBC client application that calls PARAMETER STYLE JAVA stored procedures in the `Spserver` stored procedure class.
`Spcreate.db2`	A CLP script that contains the CREATE PROCEDURE statements to register the methods contained in the Spserver class as stored procedures.
`Spdrop.db2`	A CLP script that contains the DROP PROCEDURE statements necessary for deregistering the stored procedures contained in the Spserver class.
`Spserver.java`	A JDBC program demonstrating PARAMETER STYLE JAVA stored procedures. The client program is `Spclient.java`.
`UDFcli.java`	A JDBC client application that calls functions in the Java user-defined function library, `UDFsrv`.

Table 46. Java Database Connectivity (JDBC) Sample Programs (continued)

Sample Program Name	Program Description
UseThrds.java	Shows how to use threads to run an SQL statement asynchronously (JDBC version of CLI sample async.c).
V5SpCli.java	A Java client application that calls the DB2GENERAL stored procedure, V5Stp.java.
V5Stp.java	Demonstrates a DB2GENERAL stored procedure that updates the EMPLOYEE table on the server, and returns new salary and payroll information to the client. The client program is V5SpCli.java.
Varinp.java	Demonstrates variable input to Embedded Dynamic SQL statement calls using parameter markers.

Table 47. Embedded SQL for Java (SQLJ) Sample Programs

Sample Program Name	Program Description
App.sqlj	Uses static SQL to retrieve and update data from the EMPLOYEE table of the sample database.
Applt.sqlj	An applet that queries the database using the JDBC applet driver. It uses the user name, password, server, and port number parameters specified in Applt.html.
Applt.html	An HTML file that embeds the applet sample program, Applt. It needs to be customized with server and user information.
Cursor.sqlj	Demonstrates an iterator using static SQL.
OpF_Curs.sqlj	Class file for the Openftch program.
Openftch.sqlj	Demonstrates fetching, updating, and deleting rows using static SQL.
Outsrv.sqlj	Demonstrates a stored procedure using the SQLDA structure. It fills the SQLDA with the median salary of the employees in the STAFF table of the sample database. After the database processing (finding the median), the stored procedure returns the filled SQLDA and the SQLCA status to the JDBC client application, Outcli.
Stclient.sqlj	An SQLJ client application that calls PARAMETER STYLE JAVA stored procedures created by the SQLJ stored procedure program, Stserver.
Stcreate.db2	A CLP script that contains the CREATE PROCEDURE statements to register the methods contained in the Stserver class as stored procedures.
Stdrop.db2	A CLP script that contains the DROP PROCEDURE statements necessary for deregistering the stored procedures contained in the Stserver class.
Stserver.sqlj	An SQLJ program demonstrating PARAMETER STYLE JAVA stored procedures. The client program is Stclient.sqlj.
Static.sqlj	Uses static SQL to retrieve information.
Stp.sqlj	A stored procedure that updates the EMPLOYEE table on the server, and returns new salary and payroll information to the JDBC client program, StpCli.

Table 47. Embedded SQL for Java (SQLJ) Sample Programs (continued)

Sample Program Name	Program Description
UDFclie.sqlj	A client application that calls functions from the Java user-defined function library, UDFsrv.
Updat.sqlj	Uses static SQL to update a database.

SQL Procedure Samples

Table 48. SQL Procedure Sample Programs

Sample Program Name	Program Description
basecase.db2	The UPDATE_SALARY procedure raises the salary of an employee identified by the ″empno″ IN parameter in the ″staff″ table of the ″sample″ database. The procedure determines the raise according to a CASE statement that uses the ″rating″ IN parameter.
basecase.sqc	Calls the UPDATE_SALARY procedure.
baseif.db2	The UPDATE_SALARY_IF procedure raises the salary of an employee identified by the ″empno″ IN parameter in the ″staff″ table of the ″sample″ database. The procedure determines the raise according to an IF statement that uses the ″rating″ IN parameter.
baseif.sqc	Calls the UPDATE_SALARY_IF procedure.
dynamic.db2	The CREATE_DEPT_TABLE procedure uses dynamic DDL to create a new table. The name of the table is based on the value of the IN parameter to the procedure.
dynamic.sqc	Calls the CREATE_DEPT_TABLE procedure.
iterate.db2	The ITERATOR procedure uses a FETCH loop to retrieve data from the ″department″ table. If the value of the ″deptno″ column is not ′D11′, modified data is inserted into the ″department″ table. If the value of the ″deptno″ column is ′D11′, an ITERATE statement passes the flow of control back to the beginning of the LOOP statement.
iterate.sqc	Calls the ITERATOR procedure.
leave.db2	The LEAVE_LOOP procedure counts the number of FETCH operations performed in a LOOP statement before the ″not_found″ condition handler invokes a LEAVE statement. The LEAVE statement causes the flow of control to exit the loop and complete the stored procedure.
leave.sqc	Calls the LEAVE_LOOP procedure.
loop.db2	The LOOP_UNTIL_SPACE procedure counts the number of FETCH operations performed in a LOOP statement until the cursor retrieves a row with a space (′ ′) value for column ″midinit″. The loop statement causes the flow of control to exit the loop and complete the stored procedure.
loop.sqc	Calls the LOOP_UNTIL_SPACE procedure.

Table 48. SQL Procedure Sample Programs (continued)

Sample Program Name	Program Description
nestcase.db2	The BUMP_SALARY procedure uses nested CASE statements to raise the salaries of employees in a department identified by the dept IN parameter from the ″staff″ table of the ″sample″ database.
nestcase.sqc	Calls the BUMP_SALARY procedure.
nestif.db2	The BUMP_SALARY_IF procedure uses nested IF statements to raise the salaries of employees in a department identified by the dept IN parameter from the ″staff″ table of the ″sample″ database.
nestif.sqc	Calls the BUMP_SALARY_IF procedure.
repeat.db2	The REPEAT_STMT procedure counts the number of FETCH operations performed in a repeat statement until the cursor can retrieve no more rows. The condition handler causes the flow of control to exit the repeat loop and complete the stored procedure.
repeat.sqc	Calls the REPEAT_STMT procedure.
rsultset.c	Calls the MEDIAN_RESULT_SET procedure, displays the median salary, then displays the result set generated by the SQL procedure. This client is written using the CLI API, which can accept result sets.
rsultset.db2	The MEDIAN_RESULT_SET procedure obtains the median salary of employees in a department identified by the ″dept″ IN parameter from the ″staff″ table of the ″sample″ database. The median value is assigned to the salary OUT parameter and returned to the ″rsultset″ client. The procedure then opens a WITH RETURN cursor to return a result set of the employees with a salary greater than the median. The procedure returns the result set to the client.
spserver.db2	The SQL procedures in this CLP script demonstrate basic error-handling, nested stored procedure calls, and returning result sets to the client application or the calling application. You can call the procedures using the ″spcall″ application, in the CLI samples directory. You can also use the ″spclient″ application, in the C and CPP samples directories, to call the procedures that do not return result sets.
whiles.db2	The DEPT_MEDIAN procedure obtains the median salary of employees in a department identified by the ″dept″ IN parameter from the ″staff″ table of the ″sample″ database. The median value is assigned to the salary OUT parameter and returned to the ″whiles″ client. The whiles client then prints the median salary.
whiles.sqc	Calls the DEPT_MEDIAN procedure.

ADO, RDO, and MTS Samples

Table 49. ADO, RDO, and MTS Sample Programs

Sample Program Name	Program Description
Bank.vbp	An RDO program to create and maintain data for bank branches, with the ability to perform transactions on customer accounts. The program can use any database specified by the user as it contains the DDL to create the necessary tables for the application to store data.
Blob.vbp	This ADO program demonstrates retrieving BLOB data. It retrieves and displays pictures from the emp_photo table of the sample database. The program can also replace an image in the emp_photo table with one from a local file.
BLOBAccess.dsw	This sample demonstrates highlighting ADO/Blob access using Microsoft Visual C++. It is similar to the Visual Basic sample, Blob.vbp. The BLOB sample has two main functions: 1. Read a BLOB from the Sample database and display it to the screen. 2. Read a BLOB from a file and insert it into the database. (Import)
Connect.vbp	This ADO program will create a connection object, and establish a connection, to the sample database. Once completed, the program will disconnect and exit.
Commit.vbp	This application demonstrates the use of autocommit/manual-commit features of ADO. The program queries the EMPLOYEE table of the sample database for employee number and name. The user has an option of connecting to the database in either autocommit or manual-commit mode. In the autocommit mode, all of the changes that a user makes on a record are updated automatically in the database. In the manual-commit mode, the user needs to begin a transaction before he/she can make any changes. The changes made since the beginning of a transaction can be undone by performing a rollback. The changes can be saved permanently by committing the transaction. Exiting the program automatically rolls back the changes.
db2com.vbp	This Visual Basic project demonstrates updating a database using the Microsoft Transaction Server. It creates a server DLL used by the client program, db2mts.vbp, and has four class modules: • UpdateNumberColumn.cls • UpdateRow.cls • UpdateStringColumn.cls • VerifyUpdate.cls For this program a temporary table, DB2MTS, is created in the sample database.
db2mts.vbp	This is a Visual Basic project for a client program that uses the Microsoft Transaction Server to call the server DLL created from db2com.vbp.
Select-Update.vbp	This ADO program performs the same functions as Connect.vbp, but also provides a GUI interface. With this interface, the user can view, update, and delete data stored in the ORG table of the sample database.

Table 49. ADO, RDO, and MTS Sample Programs (continued)

Sample Program Name	Program Description
Sample.vbp	This Visual Basic project uses Keyset cursors via ADO to provide a graphical user interface to all data in the sample database.
VarCHAR.dsp	A Visual C++ program that uses ADO to access VarChar data as textfields. It provides a graphical user interface to allow users to view and update data in the ORG table of the sample database.

Object Linking and Embedding Samples

Table 50. 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).
bcounter	An OLE automation user-defined function demonstrating a scratchpad using instance variables (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).
salarycltvc	A Visual C++ embedded SQL sample that calls the Visual Basic stored procedure, salarysrv.
salarycltvb	A Visual Basic DB2 CLI sample that calls the Visual Basic stored procedure, salarysrv.
testcli	An OLE automation embedded SQL client application that calls the stored procedure, tstsrv (implemented in Visual C++).
tstsrv	An OLE automation stored procedure demonstrating the passing of various types between client and stored procedure (implemented in Visual C++).

Table 51. Object Linking and Embedding Database (OLE DB) Table Functions

Sample Program Name	Program Description
jet.db2	Microsoft.Jet.OLEDB.3.51 Provider

Table 51. Object Linking and Embedding Database (OLE DB) Table Functions (continued)

Sample Program Name	Program Description
mapi.db2	INTERSOLV Connect OLE DB for MAPI
msdaora.db2	Microsoft OLE DB Provider for Oracle
msdasql.db2	Microsoft OLE DB Provider for ODBC Drivers
msidxs.db2	Microsoft OLE DB Index Server Provider
notes.db2	INTERSOLV Connect OLE DB for Notes
sampprov.db2	Microsoft OLE DB Sample Provider
sqloledb.db2	Microsoft OLE DB Provider for SQL Server

Command Line Processor Samples

Table 52. Command Line Processor (CLP) Sample Programs.

Sample File Name	File Description
const.db2	Creates a table with a CHECK CONSTRAINT clause.
cte.db2	Demonstrates a common table expression. The equivalent sample program demonstrating this advanced SQL statement is tabsql.
flt.db2	Demonstrates a recursive query. The equivalent sample program demonstrating this advanced SQL statement is recursql.
join.db2	Demonstrates an outer join of tables. The equivalent sample program demonstrating this advanced SQL statement is joinsql.
stock.db2	Demonstrates the use of triggers. The equivalent sample program demonstrating this advanced SQL statement is trigsql.
testdata.db2	Uses DB2 built-in functions such as RAND() and TRANSLATE() to populate a table with randomly generated test data.
thaisort.db2	This script is particularly for Thai users. Thai sorting is by phonetic order requiring pre-sorting/swapping of the leading vowel and its consonant, as well as post-sorting in order to view the data in the correct sort order. The file implements Thai sorting by creating UDF functions presort and postsort, and creating a table; then it calls the functions against the table to sort the table data. To run this program, you first have to build the user-defined function program, udf, from the C source file, udf.c.

Log Management User Exit Samples

Table 53. Log Management User Exit Sample Programs.

Sample File Name	File Description
db2uext2.cadsm	This is a sample User Exit utilizing ADSTAR DSM (ADSM) APIs to archive and retrieve database log files. The sample provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX and Windows 32-bit operating systems. The OS/2 version is db2uexit.cad.
db2uexit.cad	This is the OS/2 version of db2uext2.cadsm. The file must be renamed db2uexit.c and compiled as a C program.
db2uext2.cdisk	This is a sample User Exit utilizing the system copy command for the particular platform on which it ships. The program archives and retrieves database log files, and provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX and Windows 32-bit operating systems.
db2uext2.ctape	This is a sample User Exit utilizing system tape commands for the particular UNIX platform on which it ships. The program archives and retrieves database log files. All limitations of the system tape commands are limitations of this user exit. The sample provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX platforms only.

Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs

This chapter describes how you can write DB2DARI and DB2GENERAL parameter style stored procedures and DB2GENERAL UDFs.

DB2DARI Stored Procedures

When invoked, the DB2DARI 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.

Using the SQLDA in a Client Application

To use the SQLDA structure to pass values to the stored procedure, perform the following steps before calling the stored procedure:

1. Allocate storage for the structure with the required 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 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. For information on the SQLDA structure, refer to the *SQL Reference*.

Using Host Variables in a DB2DARI Client

Declare SQLVARs using the same approach discussed in "Allocating Host Variables" on page 198. In addition, the client application should set the indicator of output-only SQLVARs to -1 as discussed in "Data Structure Manipulation" on page 767. 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.

Using the SQLDA in a 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.

Note: 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.

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`.

Summary of Data Structure Usage

Table 54 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 54. 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

Table 54. Stored Procedures Parameter Variables (continued)

	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.

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 for DB2DARI Stored Procedures

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 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 improve performance.

If the stored procedure is invoked only once, it should return `SQLZ_DISCONNECT_PROC`.

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 stored procedure should return the value `SQLZ_DISCONNECT_PROC` to free the stored procedure library from main memory. Otherwise, the library remains in main memory until the database manager is stopped. As an alert to the stored procedure, the client application could pass a flag in one of the parameters indicating the final call.

DB2GENERAL UDFs and Stored Procedures

PARAMETER STYLE DB2GENERAL UDFs and stored procedures are written in Java, and hereafter are referred to simply as Java UDFs and stored procedures. Creating DB2GENERAL 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.

Supported SQL Data Types

When you call PARAMETER STYLE DB2GENERAL UDFs and stored procedures, DB2 converts SQL types to and from Java types for you as described in Table 55. Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 55. DB2 SQL Types and Java Objects

SQL Column Type	Java Type (UDF)	Java Type (Stored Procedure)
SMALLINT (500/501)	short	short
INTEGER (496/497)	int	int
BIGINT (492/493)	long	long
FLOAT (480/481)	double	double
REAL (480/481)[1]	float	float
DECIMAL(p,s) (484/485)	java.math.BigDecimal	java.math.BigDecimal
NUMERIC(p,s) (504/505)	java.math.BigDecimal	java.math.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)	String	String
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]	String	String
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

Table 55. DB2 SQL Types and Java Objects (continued)

SQL Column Type	Java Type (UDF)	Java Type (Stored Procedure)

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. See "Classes for Java Stored Procedures and UDFs" 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.

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.

Classes for Java Stored Procedures and UDFs

Java stored procedures are very similar to Java UDFs. 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 inherit from the `COM.ibm.db2.app.StoredProc` class instead of the `COM.ibm.db2.app.UDF` class. Refer to "COM.ibm.db2.app.StoredProc" on page 772 for a description of the `COM.ibm.db2.app.StoredProc` class.

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.

There are five 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.Lob
- 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 PARAMETER STYLE DB2GENERAL stored procedures must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-STP-class> extends 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. All arguments of a PARAMETER STYLE DB2GENERAL 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 PARAMETER STYLE DB2GENERAL UDFs must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-UDF-class> extends 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. For table functions, close() is called after the CLOSE call to the UDF method (if NO FINAL CALL is coded or defaulted), or after the FINAL call (if FINAL CALL is coded). If a Java UDF class does not implement this function, a no-op stub will handle and ignore this event.

```
public int getCallType() throws Exception
```

Table function UDF methods use getCallType() to find out the call type for a particular call. It returns a value as follows (symbolic defines are provided for these values in the COM.ibm.db2.app.UDF class definition):

- -2 FIRST call
- -1 OPEN call
- 0 FETCH call
- 1 CLOSE call
- 2 FINAL call

```
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
public String getDBapplid() 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.Lob

This class provides utility routines that create temporary Blob or Clob objects for computation inside user-defined functions or stored procedures.

The following methods are associated with the COM.ibm.db2.app.Lob class:

```
public static Blob newBlob() throws Exception
```

This function creates a temporary Blob. It will be implemented using a LOCATOR if possible.

```
public static Clob newClob() throws Exception
```

This function creates a temporary Clob. It will be implemented using a LOCATOR if possible.

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 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 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.

NOT FENCED Stored Procedures

To indicate that a DB2DARI stored procedure should run as a NOT FENCED stored procedure, place it in the directory indicated in the *Application Building Guide*. For more information on NOT FENCED stored procedures, see "NOT FENCED Stored Procedures" on page 231.

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 23.

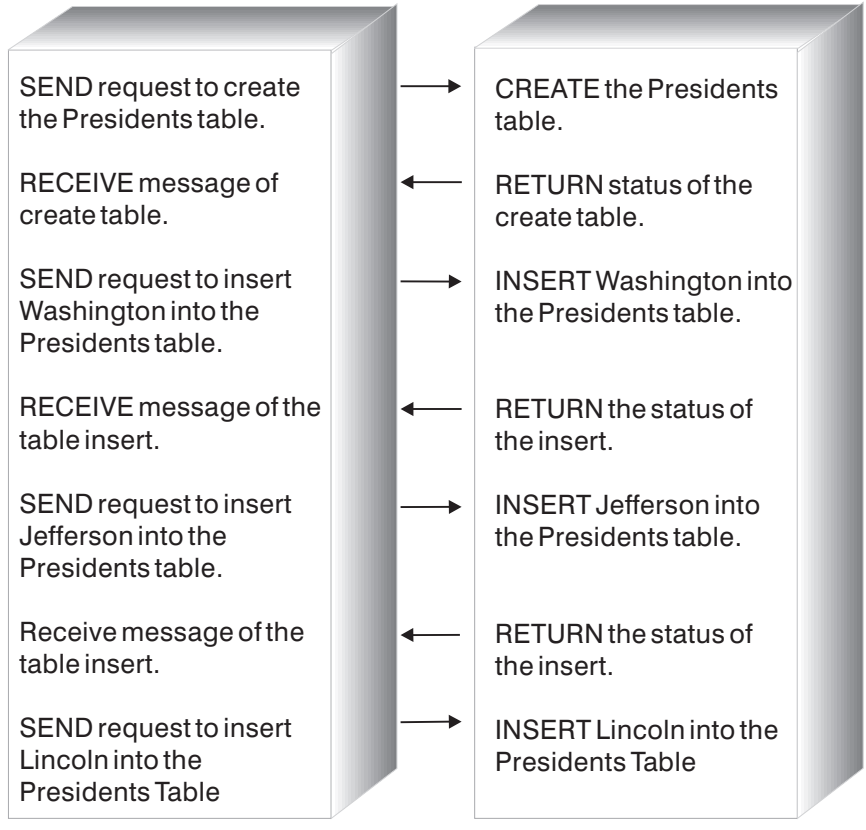


Figure 23. Input-SQLDA Sample Without a Stored Procedure

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 24.

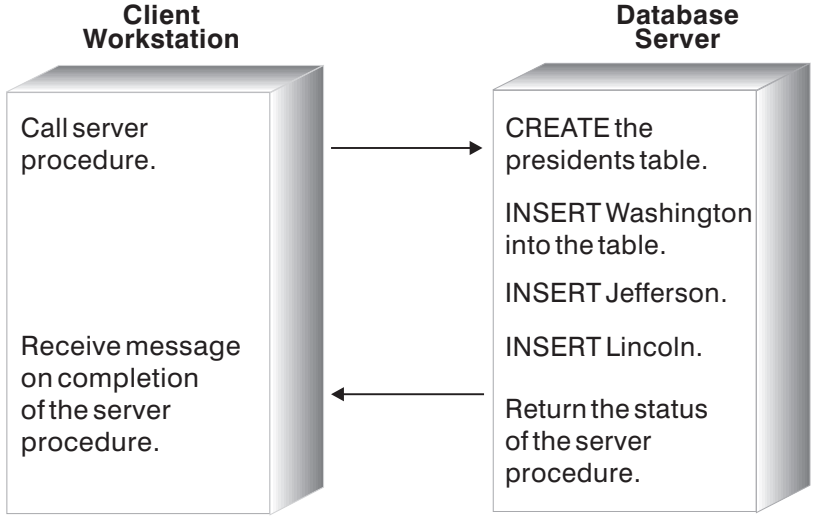


Figure 24. Input-SQLDA Sample With a Stored Procedure

A sample input-SQLDA client application and sample input-SQLDA stored procedure is shown on 779.

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 For C programs that call DB2 APIs, the `sqlInfoPrint` function in utilapi.c is redefined as `API_SQL_CHECK` in utilapi.h. For C

embedded SQL programs, the `sqlInfoPrint` function in utilemb.sqc is redefined as `EMB_SQL_CHECK` in utilemb.h.

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 119 for the source code for this error checking utility.

C Example: V5SPCLI.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;
   char eBuffer[1024]; /* error message buffer */


   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
   /* COMMIT or ROLLBACK the transaction */
```

```
if (SQLCODE == 0)
{ EXEC SQL COMMIT;
  printf("Server Procedure Complete.\n\n");
}
else
{ /* print the error message, roll back the transaction and return */
  sqlaintp (eBuffer, 1024, 80, &sqlca);
  printf("\n%s\n", eBuffer);

  EXEC SQL ROLLBACK;
  printf("Server Procedure Transaction Rolled Back.\n\n");
  return 1;
}


/* 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  = &dataind1;

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  = &dataind2;

/*********************************************\
* 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;
inout_sqlda->sqlvar[0].sqlind  = &tableind;
inout_sqlda->sqlvar[1].sqlind  = &dataind0;
inout_sqlda->sqlvar[2].sqlind  = &dataind1;
inout_sqlda->sqlvar[3].sqlind  = &dataind2;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;   2b
/* COMMIT or ROLLBACK the transaction */
if (SQLCODE == 0)
```

```
   { EXEC SQL COMMIT;
     printf("Server Procedure Complete.\n\n");
   }
   else
   { /* print the error message, roll back the transaction and return */
     sqlaintp (eBuffer, 1024, 80, &sqlca);
     printf("\n%s\n", eBuffer);

     EXEC SQL ROLLBACK;
     printf("Server Procedure Transaction Rolled Back.\n\n");
     return 1;
   }

   /* Free allocated memory */
   free( inout_sqlda );

   /* Drop the PRESIDENTS table created by the stored procedure */
   EXEC SQL DROP TABLE PRESIDENTS;
   CHECKERR("DROP TABLE");

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : inpcli.sqc */
```

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: V5SPSRV.SQC

```
#include <memory.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>

#ifdef __cplusplus
extern "C"
#endif
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))");   2

   EXEC SQL EXECUTE IMMEDIATE :table_stmt;

   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.                                  */
   /*-----------------------------------------------------------------*/

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;
     }
   }

   return(SQLZ_DISCONNECT_PROC);
}
```

Appendix D. Programming in a Distributed Environment Programming in a Host or AS/400 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, refer to the *DB2 Connect User's Guide*.

DB2 Connect lets an application program access data in DB2 databases on System/390 and AS/400 servers. For example, an application running on Windows can access data in a DB2 Universal Database for OS/390 database. You can create new applications, or modify existing applications to run in a host or AS/400 environment. You can also develop applications in one environment and port them to another.

DB2 Connect enables you to use the following APIs with host database products such as DB2 Universal Database for OS/390, as long as the item is supported by the host database product:

- Embedded SQL, both static and dynamic
- The DB2 Call Level Interface
- The Microsoft ODBC API
- JDBC.

Some SQL statements differ among relational database products. You may encounter SQL statements that are:

- The same for all the database products that you use regardless of standards
- Documented in the *SQL Reference* and are therefore available in all IBM relational database products
- 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 host or AS/400 server. For information on limits on different platforms, such as the maximum column length, refer to 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 7 database, 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 7 database, except in cases where a restriction is due to a limitation of DB2 Connect itself, for example, the non-support of Abstract Data Types.

When you program in a host or AS/400 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 host or AS/400 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 host or AS/400 server when you precompile to a host or AS/400 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 Universal Database 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.

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 Universal Database 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 host or AS/400 server may support only a subset of scalar functions for long fields; for example, DB2 Universal Database for OS/390 allows only the **LENGTH** and **SUBSTR** functions for long fields. Also, a host or AS/400 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 supported by DB2 Connect.

User Defined Types (UDTs)

Only User Defined Distinct Types are supported by DB2 Connect. Abstract Data Types are not.

ROWID Data Type

The ROWID data type is handled by DB2 Connect as VARCHAR for bit data.

64-bit Integer (BIGINT) data type

Eight byte (64-bit) integers are supported by DB2 Connect. The BIGINT internal data type is used to provide support for the cardinality of very large databases, while retaining data precision.

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 Universal Database for OS/390

ARI DB2 for VSE & VM

QSQ DB2 Universal Database 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).

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 host or AS/400 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 host or AS/400 server precompilers in the following ways:

- It makes only one pass through an application.
- 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 database manager 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 database manager configuration file for the RQRIOBLK field. Current versions of DB2 Connect support block sizes up to 32 767. If larger values are specified in the DB2 database manager configuration file, DB2 Connect uses a value of 32 767 but does not reset the DB2 database manager configuration file. Blocking is handled the same way using the same block size for dynamic and static SQL.

Note: Most host or AS/400 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 database manager configuration file by using the CLP, the Control Center, or an API, as listed in the *Administrative 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 host or AS/400 server system has limitations on the use of these attributes:

DB2 Universal Database for OS/390
All four attributes can be different. The use of a different qualifier requires special administrative privileges. For more information on the conditions concerning the usage of these attributes, refer to the *Command Reference* for DB2 Universal Database for OS/390.

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 Universal Database 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
All four attributes can be different. The use of a different owner requires administrative authority and the binder must have CREATEIN privilege on the schema (if it already exists).

Note: DB2 Connect provides support for the *SET CURRENT PACKAGESET* command for DB2 Universal Database 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 in C and C++" on page 617 for a description of how null-terminated strings are handled when prepared with the LANGLEVEL option set to MIA or SAA1.

For a description of how null-terminated strings are handled when prepared with the LANGLEVEL option set to MIA or SAA1, refer to *Application Development Guide*.

By default, CNULREQD is set to YES. This causes null-terminated strings to be interpreted according to MIA standards. If connecting to a DB2 Universal Database for OS/390 server it is strongly recommended to set CNULREQD to YES. 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, refer to the *Application Development Guide*, the *Administrative API Reference* and the *Command Reference*.

Note: Database tables can now be stored on DB2 Universal Database for OS/390 in ASCII format. This permits faster exchange of data between DB2 Connect and DB2 Universal Database 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 Universal Database 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 Universal Database 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 unique constraint, including a primary 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 Universal Database 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 host or AS/400 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 database server that does not have SQLCODE mapping (a non-IBM database server). You can also turn off SQLCODE mapping.

For more information, see .

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.

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 Universal Database 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 Universal Database 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 host or AS/400 server does not support the isolation level that you specify, the next higher supported level is used.

Table 56 shows the result of each isolation level on each host or AS/400 application server.

Table 56. Isolation Levels

DB2 Connect	DB2 Universal Database for OS/390	DB2 for VSE & VM	DB2 Universal Database 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 Universal Database for AS/400 that matches RR. DB2 Universal Database 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 Universal Database 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.

  OS/390
  The schema name must be no more than 8 bytes long, the procedure name must be no more than 18 bytes long, and the stored procedure must be defined in the SYSIBM.SYSPROCEDURES catalog on the server.

  VSE or VM
  The procedure name must not be more than 18 bytes long and must be defined in the SYSTEM.SYSROUTINES catalog on the server.

  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, refer to the *SQL Reference*. For information on how to use stored procedures when writing application programs, refer to the *Application Development Guide*.

You can invoke the server program on DB2 Universal Database with the same parameter convention that server programs use on DB2 Universal Database for OS/390, DB2 Universal Database for AS/400, or DB2 for VSE & VM. For more information on invoking DB2 Universal Database stored procedures, see "Chapter 7. Stored Procedures" on page 193refer to the *Application Development Guide*. For more information on the parameter convention on other platforms, 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 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 Universal Database for OS/390 or DB2 Universal Database for AS/400 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.
- DB2 for VSE & VM Version 7 or higher and DB2 Universal Database for OS/390 Version 5.1 or higher are the only host or AS/400 Application Servers that can return the result sets of stored procedures at this time.

Stored Procedure Builder

DB2 Stored Procedure Builder provides an easy-to-use development environment for creating, installing, and testing stored procedures. It allows you to focus on creating your stored procedure logic rather than the details of

registering, building, and installing stored procedures on a DB2 server. Additionally, with Stored Procedure Builder, you can develop stored procedures on one operating system and build them on other server operating systems.

Stored Procedure Builder is a graphical application that supports rapid development. Using Stored Procedure Builder, you can perform the following tasks:

- Create new stored procedures.
- Build stored procedures on local and remote DB2 servers.
- Modify and rebuild existing stored procedures.
- Test and debug the execution of installed stored procedures.

You can launch Stored Procedure Builder as a separate application from the DB2 Universal Database program group, or you can launch Stored Procedure Builder from any of the following development applications:

- Microsoft Visual Studio
- Microsoft Visual Basic
- IBM VisualAge for Java

You can also launch Stored Procedure Builder from Control Center for DB2 for OS/390. You can start Stored Procedure Builder as a separate process from the Control Center Tools menu, toolbar, or Stored Procedures folder. In addition, from the Stored Procedure Builder Project window, you can export one or more selected SQL stored procedures built to a DB2 for OS/390 server to a specified file capable of running within the Command Line Processor (CLP).

Stored Procedure Builder manages your work by using projects. Each Stored Procedure Builder project saves your connections to specific databases, such as DB2 for OS/390 servers. In addition, you can create filters to display subsets of the stored procedures on each database. When opening a new or existing Stored Procedure Builder project, you can filter stored procedures so that you view stored procedures based on their name, schema, language, or collection ID (for OS/390 only).

Connection information is saved in a Stored Procedure Builder project; therefore, when you open an existing project, you are automatically prompted to enter your user id and password for the database. Using the Inserting SQL Stored Procedure wizard, you can build SQL stored procedures on a DB2 for OS/390 server. For a SQL stored procedure built to a DB2 for OS/390 server, you can set specific compile, pre-link, link, bind, runtime, WLM environment, and external security options.

Additionally, you can obtain SQL costing information about the SQL stored procedure, including information about CPU time and other DB2 costing information for the thread on which the SQL stored procedure is running. In particular, you can obtain costing information about latch/lock contention wait time, the number of getpages, the number of read I/Os, and the number of write I/Os.

To obtain costing information, Stored Procedure Builder connects to a DB2 for OS/390 server, executes the SQL statement, and calls a stored procedure (DSNWSPM) to find out how much CPU time the SQL stored procedure used.

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 host or AS/400 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, refer to the *SQL Reference*.

Multisite Update with DB2 Connect

DB2 Connect allows you to perform a multisite update, also known as two-phase commit. A multisite update is an update of 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.
- If you have SNA network connections, you can use two-phase commit support provided by the sync point manager function of DB2 Connect

Enterprise Edition Version 7 on AIX, OS/2, and Windows NT. This enables the following host database servers to participate in a distributed unit of work:

– DB2 for AS/400 Version 3.1 or later
– DB2 for MVS/ESA Version 3.1 or later
– DB2 for OS/390 Version 5.1 or later
– DB2 for VM & VSE Version V5.1 or later.

The above is true for native DB2 UDB applications and applications coordinated by an external Transaction Processing (TP) Monitor such as IBM TXSeries, CICS for Open Systems, BEA Tuxedo, Encina Monitor, and Microsoft Transaction Server.

Note: For more information on BEA Tuxedo, see .For more information on the XA concentrator, see .

- If you have TCP/IP network connections, then a DB2 for OS/390 V5.1 or later server can participate in a distributed unit of work. If the application is controlled by a Transaction Processing Monitor such as IBM TXSeries, CICS for Open Systems, Encina Monitor, or Microsoft Transaction Server, then you must use the sync point manager.

  If a common DB2 Connect Enterprise Edition server is used by both native DB2 applications and TP monitor applications to access host data over TCP/IP connections then the sync point manager must be used.

  If a single DB2 Connect Enterprise Edition server is used to access host data using both SNA and TCP/IP network protocols and two phase commit is required, then the sync point manager must be used. This is true for both DB2 applications and TP monitor applications.

Host or AS/400 Server SQL Statements Supported by DB2 Connect

The following statements compile successfully for host or AS/400 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 host or AS/400 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.

Host or AS/400 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 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 26 on page 806 for the definition of code page 500, and Figure 27 on page 807 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 25 on page 804.

```
#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 25. 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 27 on page 807. 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 Sequence Overview" on page 504. Also see the CREATE DATABASE API described in the *Administrative 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 26. 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 27. Code Page 850

Appendix F. Using the DB2 Library

The DB2 Universal Database library consists of online help, books (PDF and HTML), and sample programs in HTML format. This section describes the information that is provided, and how you can access it.

To access product information online, you can use the Information Center. For more information, see "Accessing Information with the Information Center" on page 823. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web.

DB2 PDF Files and Printed Books

DB2 Information

The following table divides the DB2 books into four categories:

DB2 Guide and Reference Information
These books contain the common DB2 information for all platforms.

DB2 Installation and Configuration Information
These books are for DB2 on a specific platform. For example, there are separate *Quick Beginnings* books for DB2 on OS/2, Windows, and UNIX-based platforms.

Cross-platform sample programs in HTML
These samples are the HTML version of the sample programs that are installed with the Application Development Client. The samples are for informational purposes and do not replace the actual programs.

Release notes
These files contain late-breaking information that could not be included in the DB2 books.

The installation manuals, release notes, and tutorials are viewable in HTML directly from the product CD-ROM. Most books are available in HTML on the product CD-ROM for viewing and in Adobe Acrobat (PDF) format on the DB2 publications CD-ROM for viewing and printing. You can also order a printed copy from IBM; see "Ordering the Printed Books" on page 819. The following table lists books that can be ordered.

On OS/2 and Windows platforms, you can install the HTML files under the sqllib\doc\html directory. DB2 information is translated into different

languages; however, all the information is not translated into every language. Whenever information is not available in a specific language, the English information is provided

On UNIX platforms, you can install multiple language versions of the HTML files under the doc/%L/html directories, where %L represents the locale. For more information, refer to the appropriate *Quick Beginnings* book.

You can obtain DB2 books and access information in a variety of ways:

- "Viewing Information Online" on page 822
- "Searching Information Online" on page 826
- "Ordering the Printed Books" on page 819
- "Printing the PDF Books" on page 818

Table 57. DB2 Information

Name	Description	Form Number PDF File Name	HTML Directory
	DB2 Guide and Reference Information		
Administration Guide	*Administration Guide: Planning* provides an overview of database concepts, information about design issues (such as logical and physical database design), and a discussion of high availability.	SC09-2946 db2d1x70	db2d0
	Administration Guide: Implementation provides information on implementation issues such as implementing your design, accessing databases, auditing, backup and recovery.	SC09-2944 db2d2x70	
	Administration Guide: Performance provides information on database environment and application performance evaluation and tuning.	SC09-2945 db2d3x70	
	You can order the three volumes of the *Administration Guide* in the English language in North America using the form number SBOF-8934.		
Administrative API Reference	Describes the DB2 application programming interfaces (APIs) and data structures that you can use to manage your databases. This book also explains how to call APIs from your applications.	SC09-2947 db2b0x70	db2b0

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
Application Building Guide	Provides environment setup information and step-by-step instructions about how to compile, link, and run DB2 applications on Windows, OS/2, and UNIX-based platforms.	SC09-2948 db2axx70	db2ax
APPC, CPI-C, and SNA Sense Codes	Provides general information about APPC, CPI-C, and SNA sense codes that you may encounter when using DB2 Universal Database products. Available in HTML format only.	No form number db2apx70	db2ap
Application Development Guide	Explains how to develop applications that access DB2 databases using embedded SQL or Java (JDBC and SQLJ). Discussion topics include writing stored procedures, writing user-defined functions, creating user-defined types, using triggers, and developing applications in partitioned environments or with federated systems.	SC09-2949 db2a0x70	db2a0
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.	SC09-2950 db2l0x70	db2l0
Command Reference	Explains how to use the Command Line Processor and describes the DB2 commands that you can use to manage your database.	SC09-2951 db2n0x70	db2n0
Connectivity Supplement	Provides setup and reference information on how 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. This book also details how to use DRDA application servers with DB2 Connect application requesters. Available in HTML and PDF only.	No form number db2h1x70	db2h1

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
Data Movement Utilities Guide and Reference	Explains how to use DB2 utilities, such as import, export, load, AutoLoader, and DPROP, that facilitate the movement of data.	SC09-2955 db2dmx70	db2dm
Data Warehouse Center Administration Guide	Provides information on how to build and maintain a data warehouse using the Data Warehouse Center.	SC26-9993 db2ddx70	db2dd
Data Warehouse Center Application Integration Guide	Provides information to help programmers integrate applications with the Data Warehouse Center and with the Information Catalog Manager.	SC26-9994 db2adx70	db2ad
DB2 Connect User's Guide	Provides concepts, programming, and general usage information for the DB2 Connect products.	SC09-2954 db2c0x70	db2c0
DB2 Query Patroller Administration Guide	Provides an operational overview of the DB2 Query Patroller system, specific operational and administrative information, and task information for the administrative graphical user interface utilities.	SC09-2958 db2dwx70	db2dw
DB2 Query Patroller User's Guide	Describes how to use the tools and functions of the DB2 Query Patroller.	SC09-2960 db2wwx70	db2ww
Glossary	Provides definitions for terms used in DB2 and its components. Available in HTML format and in the *SQL Reference*.	No form number db2t0x70	db2t0
Image, Audio, and Video Extenders Administration and Programming	Provides general information about DB2 extenders, and information on the administration and configuration of the image, audio, and video (IAV) extenders and on programming using the IAV extenders. It includes reference information, diagnostic information (with messages), and samples.	SC26-9929 dmbu7x70	dmbu7
Information Catalog Manager Administration Guide	Provides guidance on managing information catalogs.	SC26-9995 db2dix70	db2di

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
Information Catalog Manager Programming Guide and Reference	Provides definitions for the architected interfaces for the Information Catalog Manager.	SC26-9997 db2bix70	db2bi
Information Catalog Manager User's Guide	Provides information on using the Information Catalog Manager user interface.	SC26-9996 db2aix70	db2ai
Installation and Configuration Supplement	Guides you through the planning, installation, and setup of platform-specific DB2 clients. This supplement also contains information on binding, setting up client and server communications, DB2 GUI tools, DRDA AS, distributed installation, the configuration of distributed requests, and accessing heterogeneous data sources.	GC09-2957 db2iyx70	db2iy
Message Reference	Lists messages and codes issued by DB2, the Information Catalog Manager, and the Data Warehouse Center, and describes the actions you should take. You can order both volumes of the Message Reference in the English language in North America with the form number SBOF-8932.	Volume 1 GC09-2978 db2m1x70 Volume 2 GC09-2979 db2m2x70	db2m0
OLAP Integration Server Administration Guide	Explains how to use the Administration Manager component of the OLAP Integration Server.	SC27-0787 db2dpx70	n/a
OLAP Integration Server Metaoutline User's Guide	Explains how to create and populate OLAP metaoutlines using the standard OLAP Metaoutline interface (not by using the Metaoutline Assistant).	SC27-0784 db2upx70	n/a
OLAP Integration Server Model User's Guide	Explains how to create OLAP models using the standard OLAP Model Interface (not by using the Model Assistant).	SC27-0783 db2lpx70	n/a
OLAP Setup and User's Guide	Provides configuration and setup information for the OLAP Starter Kit.	SC27-0702 db2ipx70	db2ip
OLAP Spreadsheet Add-in User's Guide for Excel	Describes how to use the Excel spreadsheet program to analyze OLAP data.	SC27-0786 db2epx70	db2ep

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
OLAP Spreadsheet Add-in User's Guide for Lotus 1-2-3	Describes how to use the Lotus 1-2-3 spreadsheet program to analyze OLAP data.	SC27-0785 db2tpx70	db2tp
Replication Guide and Reference	Provides planning, configuration, administration, and usage information for the IBM Replication tools supplied with DB2.	SC26-9920 db2e0x70	db2e0
Spatial Extender User's Guide and Reference	Provides information about installing, configuring, administering, programming, and troubleshooting the Spatial Extender. Also provides significant descriptions of spatial data concepts and provides reference information (messages and SQL) specific to the Spatial Extender.	SC27-0701 db2sbx70	db2sb
SQL Getting Started	Introduces SQL concepts and provides examples for many constructs and tasks.	SC09-2973 db2y0x70	db2y0
SQL Reference, Volume 1 and Volume 2	Describes SQL syntax, semantics, and the rules of the language. This book also includes information about release-to-release incompatibilities, product limits, and catalog views. You can order both volumes of the *SQL Reference* in the English language in North America with the form number SBOF-8933.	Volume 1 SC09-2974 db2s1x70 Volume 2 SC09-2975 db2s2x70	db2s0
System Monitor Guide and Reference	Describes how to collect different kinds of information about databases and the database manager. This book explains how to use the information to understand database activity, improve performance, and determine the cause of problems.	SC09-2956 db2f0x70	db2f0
Text Extender Administration and Programming	Provides general information about DB2 extenders and information on the administration and configuring of the text extender and on programming using the text extenders. It includes reference information, diagnostic information (with messages) and samples.	SC26-9930 desu9x70	desu9

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
Troubleshooting Guide	Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service.	GC09-2850 db2p0x70	db2p0
What's New	Describes the new features, functions, and enhancements in DB2 Universal Database, Version 7.	SC09-2976 db2q0x70	db2q0
DB2 Installation and Configuration Information			
DB2 Connect Enterprise Edition for OS/2 and Windows Quick Beginnings	Provides planning, migration, installation, and configuration information for DB2 Connect Enterprise Edition on the OS/2 and Windows 32-bit operating systems. This book also contains installation and setup information for many supported clients.	GC09-2953 db2c6x70	db2c6
DB2 Connect Enterprise Edition for UNIX Quick Beginnings	Provides planning, migration, installation, configuration, and task information for DB2 Connect Enterprise Edition on UNIX-based platforms. This book also contains installation and setup information for many supported clients.	GC09-2952 db2cyx70	db2cy
DB2 Connect Personal Edition Quick Beginnings	Provides planning, migration, installation, configuration, and task information for DB2 Connect Personal Edition on the OS/2 and Windows 32-bit operating systems. This book also contains installation and setup information for all supported clients.	GC09-2967 db2c1x70	db2c1
DB2 Connect Personal Edition Quick Beginnings for Linux	Provides planning, installation, migration, and configuration information for DB2 Connect Personal Edition on all supported Linux distributions.	GC09-2962 db2c4x70	db2c4
DB2 Data Links Manager Quick Beginnings	Provides planning, installation, configuration, and task information for DB2 Data Links Manager for AIX and Windows 32-bit operating systems.	GC09-2966 db2z6x70	db2z6

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
DB2 Enterprise - Extended Edition for UNIX Quick Beginnings	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition on UNIX-based platforms. This book also contains installation and setup information for many supported clients.	GC09-2964 db2v3x70	db2v3
DB2 Enterprise - Extended Edition for Windows Quick Beginnings	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for Windows 32-bit operating systems. This book also contains installation and setup information for many supported clients.	GC09-2963 db2v6x70	db2v6
DB2 for OS/2 Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on the OS/2 operating system. This book also contains installation and setup information for many supported clients.	GC09-2968 db2i2x70	db2i2
DB2 for UNIX Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on UNIX-based platforms. This book also contains installation and setup information for many supported clients.	GC09-2970 db2ixx70	db2ix
DB2 for Windows Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database on Windows 32-bit operating systems. This book also contains installation and setup information for many supported clients.	GC09-2971 db2i6x70	db2i6
DB2 Personal Edition Quick Beginnings	Provides planning, installation, migration, and configuration information for DB2 Universal Database Personal Edition on the OS/2 and Windows 32-bit operating systems.	GC09-2969 db2i1x70	db2i1
DB2 Personal Edition Quick Beginnings for Linux	Provides planning, installation, migration, and configuration information for DB2 Universal Database Personal Edition on all supported Linux distributions.	GC09-2972 db2i4x70	db2i4

Table 57. DB2 Information (continued)

Name	Description	Form Number PDF File Name	HTML Directory
DB2 Query Patroller Installation Guide	Provides installation information about DB2 Query Patroller.	GC09-2959 db2iwx70	db2iw
DB2 Warehouse Manager Installation Guide	Provides installation information for warehouse agents, warehouse transformers, and the Information Catalog Manager.	GC26-9998 db2idx70	db2id
Cross-Platform Sample Programs in HTML			
Sample programs in HTML	Provides the sample programs in HTML format for the programming languages on all platforms supported by DB2. The sample programs are provided for informational purposes only. Not all samples are available in all programming languages. The HTML samples are only available when the DB2 Application Development Client is installed. For more information on the programs, refer to the *Application Building Guide*.	No form number	db2hs
Release Notes			
DB2 Connect Release Notes	Provides late-breaking information that could not be included in the DB2 Connect books.	See note #2.	db2cr
DB2 Installation Notes	Provides late-breaking installation-specific information that could not be included in the DB2 books.	Available on product CD-ROM only.	
DB2 Release Notes	Provides late-breaking information about all DB2 products and features that could not be included in the DB2 books.	See note #2.	db2ir

Notes:

1. The character *x* in the sixth position of the file name indicates the language version of a book. For example, the file name `db2d0e70` identifies the English version of the *Administration Guide* and the file name `db2d0f70` identifies the French version of the same book. The following letters are used in the sixth position of the file name to indicate the language version:

Language	Identifier
Brazilian Portuguese	b

Bulgarian	u
Czech	x
Danish	d
Dutch	q
English	e
Finnish	y
French	f
German	g
Greek	a
Hungarian	h
Italian	i
Japanese	j
Korean	k
Norwegian	n
Polish	p
Portuguese	v
Russian	r
Simp. Chinese	c
Slovenian	l
Spanish	z
Swedish	s
Trad. Chinese	t
Turkish	m

2. Late breaking information that could not be included in the DB2 books is available in the Release Notes in HTML format and as an ASCII file. The HTML version is available from the Information Center and on the product CD-ROMs. To view the ASCII file:
   - On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/*%L* directory, where *%L* represents the locale name and `DB2DIR` represents:
     - /usr/lpp/db2_07_01 on AIX
     - /opt/IBMdb2/V7.1 on HP-UX, PTX, Solaris, and Silicon Graphics IRIX
     - /usr/IBMdb2/V7.1 on Linux.
   - On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed. On OS/2 platforms, you can also double-click the **IBM DB2** folder and then double-click the **Release Notes** icon.

Printing the PDF Books

If you prefer to have printed copies of the books, you can print the PDF files found on the DB2 publications CD-ROM. Using the Adobe Acrobat Reader, you can print either the entire book or a specific range of pages. For the file name of each book in the library, see Table 57 on page 810.

You can obtain the latest version of the Adobe Acrobat Reader from the Adobe Web site at http://www.adobe.com.

The PDF files are included on the DB2 publications CD-ROM with a file extension of PDF. To access the PDF files:

1. Insert the DB2 publications CD-ROM. On UNIX-based platforms, mount the DB2 publications CD-ROM. Refer to your *Quick Beginnings* book for the mounting procedures.
2. Start the Acrobat Reader.
3. Open the desired PDF file from one of the following locations:
   - On OS/2 and Windows platforms:

     x:\doc*language* directory, where *x* represents the CD-ROM drive and *language* represent the two-character country code that represents your language (for example, EN for English).
   - On UNIX-based platforms:

     /cdrom/doc/*%L* directory on the CD-ROM, where */cdrom* represents the mount point of the CD-ROM and *%L* represents the name of the desired locale.

You can also copy the PDF files from the CD-ROM to a local or network drive and read them from there.

Ordering the Printed Books

You can order the printed DB2 books either individually or as a set (in North America only) by using a sold bill of forms (SBOF) number. To order books, contact your IBM authorized dealer or marketing representative, or phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada. You can also order the books from the Publications Web page at http://www.elink.ibmlink.ibm.com/pbl/pbl.

Two sets of books are available. SBOF-8935 provides reference and usage information for the DB2 Warehouse Manager. SBOF-8931 provides reference and usage information for all other DB2 Universal Database products and features. The contents of each SBOF are listed in the following table:

Table 58. Ordering the printed books

SBOF Number	Books Included	
SBOF-8931	• Administration Guide: Planning • Administration Guide: Implementation • Administration Guide: Performance • Administrative API Reference • Application Building Guide • Application Development Guide • CLI Guide and Reference • Command Reference • Data Movement Utilities Guide and Reference • Data Warehouse Center Administration Guide • Data Warehouse Center Application Integration Guide • DB2 Connect User's Guide • Installation and Configuration Supplement • Image, Audio, and Video Extenders Administration and Programming • Message Reference, Volumes 1 and 2	• OLAP Integration Server Administration Guide • OLAP Integration Server Metaoutline User's Guide • OLAP Integration Server Model User's Guide • OLAP Integration Server User's Guide • OLAP Setup and User's Guide • OLAP Spreadsheet Add-in User's Guide for Excel • OLAP Spreadsheet Add-in User's Guide for Lotus 1-2-3 • Replication Guide and Reference • Spatial Extender Administration and Programming Guide • SQL Getting Started • SQL Reference, Volumes 1 and 2 • System Monitor Guide and Reference • Text Extender Administration and Programming • Troubleshooting Guide • What's New
SBOF-8935	• Information Catalog Manager Administration Guide • Information Catalog Manager User's Guide • Information Catalog Manager Programming Guide and Reference	• Query Patroller Administration Guide • Query Patroller User's Guide

DB2 Online Documentation

Accessing 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* represents a keyword or the entire command. For example, `? catalog` displays help for all the CATALOG commands, while `? catalog database` displays help for the CATALOG DATABASE command.
Client Configuration Assistant Help *Command Center Help* *Control Center Help* *Data Warehouse Center Help* *Event Analyzer Help* *Information Catalog Manager Help* *Satellite Administration Center Help* *Script Center Help*	Explains the tasks you can perform in a window or notebook. The help includes overview and prerequisite information you need to know, and it describes how to use the window or notebook controls.	From a window or notebook, click the **Help** push button or press the **F1** key.

Message Help	Describes the cause of a message and any action you should take.	From the command line processor in interactive mode, enter:
		`? XXXnnnnn`
		where *XXXnnnnn* represents a valid message identifier.
		For example, `? SQL30081` displays help about the SQL30081 message.
		To view message help one screen at a time, enter:
		`? XXXnnnnn | more`
		To save message help in a file, enter:
		`? XXXnnnnn > filename.ext`
		where *filename.ext* represents the file where you want to save the message help.

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* represents an SQL statement. For example, help SELECT displays help about the SELECT statement. **Note:** SQL help is not available on UNIX-based platforms.
SQLSTATE Help	Explains SQL states and class codes.	From the command line processor in interactive mode, enter: `? sqlstate or ? class code` where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state. For example, `? 08003` displays help for the 08003 SQL state, while `? 08` displays help for the 08 class code.

Viewing Information Online

The books included with this product are in Hypertext Markup Language (HTML) softcopy format. Softcopy format enables you to search or browse the information and provides hypertext links to related information. It also makes it easier to share the library across your site.

You can view the online books or sample programs with any browser that conforms to HTML Version 3.2 specifications.

To view online books or sample programs:

- If you are running DB2 administration tools, use the Information Center.
- From a browser, click **File —>Open Page**. The page you open contains descriptions of and links to DB2 information:
  - On UNIX-based platforms, open the following page:

    `INSTHOME/sqllib/doc/%L/html/index.htm`

    where *%L* represents the locale name.
  - On other platforms, open the following page:

    `sqllib\doc\html\index.htm`

    The path is located on the drive where DB2 is installed.

If you have not installed the Information Center, you can open the page by double-clicking the **DB2 Information** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

Installing the Netscape Browser

If you do not already have a Web browser installed, you can install Netscape from the Netscape CD-ROM found in the product boxes. For detailed instructions on how to install it, perform the following:

1. Insert the Netscape CD-ROM.
2. On UNIX-based platforms only, mount the CD-ROM. Refer to your *Quick Beginnings* book for the mounting procedures.
3. For installation instructions, refer to the CDNAV*nn*.txt file, where *nn* represents your two character language identifier. The file is located at the root directory of the CD-ROM.

Accessing Information with the Information Center

The Information Center provides quick access to DB2 product information. The Information Center is available on all platforms on which the DB2 administration tools are available.

You can open the Information Center by double-clicking the Information Center icon. Depending on the system you are using, the icon is in the Information folder in the main product folder or the Windows **Start** menu.

You can also access the Information Center by using the toolbar and the **Help** menu on the DB2 Windows platform.

The Information Center provides six types of information. Click the appropriate tab to look at the topics provided for that type.

Tasks Key tasks you can perform using DB2.

Reference DB2 reference information, such as keywords, commands, and APIs.

Books DB2 books.

Troubleshooting
Categories of error messages and their recovery actions.

Sample Programs
Sample programs that come with the DB2 Application Development Client. If you did not install the DB2 Application Development Client, this tab is not displayed.

Web 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 a find feature, so you can look for a specific topic without browsing the lists.

For a full text search, follow the hypertext link in the Information Center to the **Search DB2 Online Information** search form.

The HTML search server is usually started automatically. If a search in the HTML information does not work, you may have to start the search server using one of the following methods:

On Windows
Click **Start** and select **Programs —> IBM DB2 —> Information —> Start HTML Search Server**.

On OS/2
Double-click the **DB2 for OS/2** folder, and then double-click the **Start HTML Search Server** icon.

Refer to the release notes if you experience any other problems when searching the HTML information.

Note: The Search function is not available in the Linux, PTX, and Silicon Graphics IRIX environments.

Using DB2 Wizards

Wizards help you complete specific administration tasks by taking you through each task one step at a time. Wizards are available through the Control Center and the Client Configuration Assistant. The following table lists the wizards and describes their purpose.

Note: The Create Database, Create Index, Configure Multisite Update, and Performance Configuration wizards are available for the partitioned database environment.

Wizard	Helps You to...	How to Access...
Add Database	Catalog a database on a client workstation.	From the Client Configuration Assistant, click **Add**.
Backup Database	Determine, create, and schedule a backup plan.	From the Control Center, right-click the database you want to back up and select **Backup —> Database Using Wizard**.

Wizard	Helps You to...	How to Access...
Configure Multisite Update	Configure a multisite update, a distributed transaction, or a two-phase commit.	From the Control Center, right-click the **Databases** folder and select **Multisite Update**.
Create Database	Create a database, and perform some basic configuration tasks.	From the Control Center, right-click the **Databases** folder and select **Create —> Database Using Wizard**.
Create Table	Select basic data types, and create a primary key for the table.	From the Control Center, right-click the **Tables** icon and select **Create —> Table Using Wizard**.
Create Table Space	Create a new table space.	From the Control Center, right-click the **Table Spaces** icon and select **Create —> Table Space Using Wizard**.
Create Index	Advise which indexes to create and drop for all your queries.	From the Control Center, right-click the **Index** icon and select **Create —> Index Using Wizard**.
Performance Configuration	Tune the performance of a database by updating configuration parameters to match your business requirements.	From the Control Center, right-click the database you want to tune and select **Configure Performance Using Wizard**. For the partitioned database environment, from the Database Partitions view, right-click the first database partition you want to tune and select **Configure Performance Using Wizard**.
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, right-click the database you want to restore and select **Restore —> Database Using Wizard**.

Setting Up a Document Server

By default, the DB2 information is installed on your local system. This means that each person who needs access to the DB2 information must install the same files. To have the DB2 information stored in a single location, perform the following steps:

1. Copy all files and subdirectories from \sqllib\doc\html on your local system to a Web server. Each book has its own subdirectory that contains all the necessary HTML and GIF files that make up the book. Ensure that the directory structure remains the same.

2. Configure the Web server to look for the files in the new location. For information, refer to the NetQuestion Appendix in the *Installation and Configuration Supplement*.
3. If you are using the Java version of the Information Center, you can specify a base URL for all HTML files. You should use the URL for the list of books.
4. When you are able to view the book files, you can bookmark commonly viewed topics. You will probably want to bookmark the following pages:
   - List of books
   - Tables of contents of frequently used books
   - Frequently referenced articles, such as the ALTER TABLE topic
   - The Search form

For information about how you can serve the DB2 Universal Database online documentation files from a central machine, refer to the NetQuestion Appendix in the *Installation and Configuration Supplement*.

Searching Information Online

To find information in the HTML files, use one of the following methods:

- Click **Search** in the top frame. Use the search form to find a specific topic. This function is not available in the Linux, PTX, or Silicon Graphics IRIX environments.
- Click **Index** in the top frame. Use the index to find a specific topic in the book.
- Display the table of contents or index of the help or 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. See "Accessing Information with the Information Center" on page 823 for details.

Appendix G. Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
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.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information 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.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms, which may be denoted by an asterisk(*), are trademarks of International Business Machines Corporation in the United States, other countries, or both.

ACF/VTAM
AISPO
AIX
AIX/6000
AIXwindows
AnyNet
APPN
AS/400
BookManager
CICS
C Set++
C/370
DATABASE 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eNetwork
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
LAN DistanceMVS
MVS/ESA
MVS/XA
Net.Data
OS/2
OS/390
OS/400
PowerPC
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/DS
SQL/400
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WIN-OS/2

The following terms are trademarks or registered trademarks of other companies:

Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

Java or all Java-based trademarks and logos, and Solaris are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries or both and is 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

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

Contacting IBM

If you have a technical problem, please review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. This guide suggests information that you can gather to help DB2 Customer Support 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.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-237-5511 for customer support
- 1-888-426-4343 to learn about available service options

Product Information

If you live in the U.S.A., then you can call one of the following numbers:

- 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.

http://www.ibm.com/software/data/
The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more.

http://www.ibm.com/software/data/db2/library/
The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information.

Note: This information may be in English only.

http://www.elink.ibmlink.ibm.com/pbl/pbl/
The International Publications ordering Web site provides information on how to order books.

http://www.ibm.com/education/certify/
The Professional Certification Program from the IBM Web site provides certification test information for a variety of IBM products, including DB2.

ftp.software.ibm.com
Log on as anonymous. In the directory `/ps/products/db2`, you can find demos, fixes, information, and tools relating to DB2 and many other products.

comp.databases.ibm-db2, bit.listserv.db2-l
These Internet newsgroups are available for users to discuss their experiences with DB2 products.

On Compuserve: GO IBMDB2
Enter this command to access the IBM DB2 Family forums. All DB2 products are supported through these forums.

For information on how to contact IBM outside of the United States, refer to Appendix A of the *IBM Software Support Handbook*. To access this document, go to the following Web page: http://www.ibm.com/support/, and then select the IBM Software Support Handbook link near the bottom of the page.

Note: In some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2949-01