IBM® DB2 Universal Database™

Application Development Guide: Programming Server Applications

Version 8

SC09-4827-00

IBM® DB2 Universal Database™

Application Development Guide: Programming Server Applications

Version 8

SC09-4827-00

Contents

About This Book

The *Application Development Guide* is a three-volume book that describes what you need to know about coding, debugging, building, and running DB2 applications:

- *Application Development Guide: Programming Client Applications* contains what you need to know to code standalone DB2 applications that run on DB2 clients. It includes information on:
  - Programming interfaces that are supported by DB2. High-level descriptions are provided for DB2 Developer's Edition, supported programming interfaces, facilities for creating Web applications, and DB2-provided programming features, such as routines and triggers.
  - The general structure that a DB2 application should follow. Recommendations are provided on how to maintain data values and relationships in the database, authorization considerations are described, and information is provided on how to test and debug your application.
  - Embedded SQL, both dynamic and static. The general considerations for embedded SQL are described, as well as the specific issues that apply to the usage of static and dynamic SQL in DB2 applications.
  - Supported host and interpreted languages, such as C/C++, COBOL, Perl, and REXX, and how to use embedded SQL in applications that are written in these languages.
  - Java (both JDBC and SQLj), and considerations for building Java applications that use WebSphere Application Servers.
  - The IBM OLE DB Provider for DB2 Servers. General information is provided about IBM OLE DB Provider support for OLE DB services, components, and properties. Specific information is also provided about Visual Basic and Visual C++ applications that use the OLE DB interface for ActiveX Data Objects (ADO).
  - National language support issues. General topics, such as collating sequences, the derivation of code pages and locales, and character conversions are described. More specific issues such as DBCS code pages, EUC character sets, and issues that apply in Japanese and Traditional Chinese EUC and UCS-2 environments are also described.
  - Transaction management. Issues that apply to applications that perform multisite updates, and to applications that perform concurrent transactions, are described.
  - Applications in partitioned database environments. Directed DSS, local bypass, buffered inserts, and troubleshooting applications in partitioned database environments are described.

  – Commonly used application techniques. Information is provided on how to use generated and identity columns, declared temporary tables, and how to use savepoints to manage transactions.
  – The SQL statements that are supported for use in embedded SQL applications.
  – Applications that access host and iSeries environments. The issues that pertain to embedded SQL applications that access host and iSeries envirionments are described.
  – The simulation of EBCDIC binary collation.
- *Application Development Guide: Programming Server Applications* contains what you need to know for server-side programming including routines, large objects, user-defined types, and triggers. It includes information on:
  – Routines (stored procedures, user-defined functions, and methods), including:
    - Routine performance, security, library management considerations, and restrictions.
    - Registering and writing routines, including the CREATE statements and debugging.
    - Procedure parameter modes and parameter handling.
    - Procedure result sets.
    - UDF features including scratchpads and scalar and table functions.
    - SQL procedures including debugging, and condition handling.
    - Parameter styles, authorizations, and binding of external routines.
    - Language-specific considerations for C, Java, and OLE automation routines.
    - Invoking routines
    - Function selection and passing distinct types and LOBs to functions.
    - Code pages and routines.
  – Large objects, including LOB usage and locators, reference variables, and CLOB data.
  – User-defined distinct types, including strong typing, defining and dropping UDTs, creating tables with structured types, using distinct types and typed tables for specific applications, manipulating distinct types and casting between them, and performing comparisons and assignments with distinct types, including UNION operations on distinctly typed columns.
  – User-defined structured types, including storing instances and instantiation, structured type hierarchies, defining structured type behavior, the dynamic dispatch of methods, the comparison, casting, and constructor functions, and mutator and observer methods for structured types.

- Typed tables, including creating, dropping, substituting, storing objects, defining system-generated object identifiers, and constraints on object identifier columns.
- Reference types, including relationships between objects in typed tables, semantic relationships with references, and referential integrity versus scoped references.
- Typed tables and typed views, including structured types as column types, transform functions and transform groups, host language program mappings, and structured type host variables.
- Triggers, including INSERT, UPDATE, and DELETE triggers, interactions with referential constraints, creation guidelines, granularity, activation time, transition variables and tables, triggered actions, multiple triggers, and synergy between triggers, constraints, and routines.

• *Application Development Guide: Building and Running Applications* contains what you need to know to build and run DB2 applications on the operating systems supported by DB2:
  - AIX
  - HP-UX
  - Linux
  - Solaris
  - Windows

  It includes information on:
  - How to set up your application development environment, including specific instructions for Java and SQL procedures, how to set up the sample database, and how to migrate your applications from previous versions of DB2.
  - DB2 supported servers and software to build applications, including supported compilers and interpreters.
  - The DB2 sample program files, makefiles, build files, and error-checking utility files.
  - How to build and run Java applets, applications, and routines.
  - How to build and run SQL procedures.
  - How to build and run C/C++ applications and routines.
  - How to build and run IBM and Micro Focus COBOL applications and routines.
  - How to build and run REXX applications on AIX and Windows.
  - How to build and run applications with ActiveX Data Objects (ADO) using Visual Basic and Visual C++ on Windows.
  - How to build and run applications with remote data objects using Visual C++ on Windows.

Part 1. Routines (Stored Procedures, UDFs, and Methods)

Chapter 1. Introducing Routines

Routines (Stored Procedures, UDFs, Methods)

Routines are composed of application logic that resides on the database server and can be invoked from a client or other routines. Because routines are database objects, you must register them with a database in order to invoke them.

There are three types of routines that you can implement: stored procedures, user-defined functions (UDFs), and methods. These routine types share much in common with regards to their development, but fulfill different requirements when interacting with databases. Stored procedures serve as extensions to clients and run on the database server. UDFs enable you to extend and customize SQL. Methods provide behavior for structured types.

There are two principal styles of routines:

SQL routines
SQL routines are composed entirely of SQL statements. You specify these statements in the CREATE statement that you use to register the routine.

External routines
External routines can be written in the following programming languages: C, C++, Java, and OLE. In addition to these languages, stored procedures can also be written in COBOL. Regardless of the routine type or programming language, you can include SQL statements in external routines.

An application or a routine may invoke any routine, regardless of whether the invoker and invokee are written in the same programming language. For example, C applications can invoke Java™ routines, or Java UDFs can invoke SQL procedures. Whether the languages of the application and the routine are the same or differ, DB2® transparently passes the values between the application and the routine. For more information on choosing a programming language in which to write your routine, see the topic ″Supported Routine Programming Languages″.

For assistance in developing routines, use the DB2 Development Center. It provides simple interfaces and a set of wizards that help make it easy to perform your development tasks. You can also integrate the DB2 Development Center with popular application development tools, such as Microsoft® Visual Studio.

The development of routines involves the following tasks:

1. Register the Routine. This task can occur at any time before you invoke the routine, except in the following circumstances:
   - Java routines that are catalogued on JAR files must be written before they can be registered.
   - Routines that issue SQL statements that refer to themselves must be registered before they are precompiled and bound. This also applies to situations where there is a cycle of references, for example, Routine A references Routine B, which references Routine A.
2. Write the routine.
3. Build (precompile -- for routines with embedded SQL, compile, and link) the routine. (See the related links for platform and language-specific build information.)
4. Debug and test the routine.
5. Invoke the routine.

Related concepts:

- "Routines: Stored Procedures" on page 7
- "Supported Routine Programming Languages" on page 13
- "Routines: Scalar User-Defined Functions" on page 8
- "Routines: Methods" on page 10
- "Routines: Table User-Defined Functions" on page 9

Related tasks:

- "Building JDBC Routines" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ Routines" in the *Application Development Guide: Building and Running Applications*
- "Building C Routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building IBM COBOL Routines on AIX" in the *Application Development Guide: Building and Running Applications*

- “Building Micro Focus COBOL Routines on AIX” in the *Application Development Guide: Building and Running Applications*
- “Building C/C++ Routines on Windows” in the *Application Development Guide: Building and Running Applications*
- “Building IBM COBOL Routines on Windows” in the *Application Development Guide: Building and Running Applications*
- “Building Micro Focus COBOL Routines on Windows” in the *Application Development Guide: Building and Running Applications*
- “Building Micro Focus COBOL Routines on HP-UX” in the *Application Development Guide: Building and Running Applications*
- “Building Micro Focus COBOL Routines on Solaris” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on HP-UX” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on HP-UX” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on Linux” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on Linux” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on Solaris” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on Solaris” in the *Application Development Guide: Building and Running Applications*
- “Writing Routines” on page 29
- “Registering Routines” on page 27
- “Invoking Routines” on page 145
- “Debugging Routines” on page 31

Benefits of Routines

When faced with the task of developing new functionality to interact with a database, there are two approaches you can choose from. You can add the new logic to a client application, or you can develop a routine, where the new logic will reside on the database server. There are a number of benefits in choosing the latter approach.

The following benefits can be gained by moving application logic into routines:

Reduce network traffic
When a client references a routine for invocation, no data needs to be

transmitted back to the client for processing. This is because the routine runs on the server, and all of the routine's processing is done there. Only those records that are actually required at the client need to be sent, such as output values, or stored procedure result sets. This can reduce the volume of data transmitted between the client and the database server, which, in turn, can result in improved performance.

Alleviate the processing load on the client
In environments where client performance is a concern, routines are practical means of running application logic. The client simply invokes the routine, and the processing is done on the database server.

Allow faster, more efficient execution
Because routines are database objects, they can have a closer relationship with the database manager than clients do, resulting in better performance if the application logic contains SQL. NOT FENCED routines run in the same process as the database manager. FENCED routines use shared memory for communication, making them more proficient in transmitting SQL requests and data than protocols such as TCP/IP, which are commonly used by clients.

Enable controlled access to database objects
You can use routines to control access to database objects. While users may not have permission to issue certain statements, they may have permission to run routines that do issue them.

Facilitate the encapsulation of application logic
In an environment where there are numerous clients, all with their own applications, the effective use of routines can simplify code reuse, standardization, and maintenance. For example, if a particular aspect of application behavior needs to be changed in an environment where routines are used, only the affected routine is modified. If routines had not been used in this instance, application logic changes would have been required on each client's system.

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Performance Considerations for Developing Routines" on page 16
- "Security Considerations for Routines" on page 20

Types of Routines

There are three types of routines you can develop: stored procedures, user-defined functions (UDFs), and methods. While the details involved in writing and registering them are similar, they each lend themselves to different uses.

The following sections present the features of each routine type in a format that facilitates comparison. Note that there are two sections for UDFs: scalar UDFs and table UDFs. They are sufficiently distinct as to warrant individual attention.

Routines: Stored Procedures

A stored procedure serves as an extension to clients that runs on the database server. They can be invoked from a client application or another routine with a CALL statement. Stored procedures and their calling programs exchange data using parameters defined in the CREATE PROCEDURE statement. Stored procedures can also return result sets to their callers.

Benefits

- Enable multiple SQL statements to be issued by a single invocation from the caller, thus minimizing data transfer between the client and the database server. The more SQL statements you include in a stored procedure, the lower the data transfer costs for each individual statement, as compared to issuing the same statements from the client. Note that if only one SQL statement is invoked in a stored procedure, the overhead in setting up this invocation may outweigh the benefit in network traffic savings.
- Isolate database logic from application logic.
- Can return multiple result sets.
- If invoked from an application, they behave as part of the application.

Limitations

- Cannot be invoked as part of any SQL statement except CALL.
- Results cannot be directly used by another SQL statement.
- Cannot preserve state between invocations.

Common uses

- Provide a common interface to a group of SQL statements. For example, given a new hire, insert rows into the employee, address, and department tables.
- Standardize application logic.

Supported languages

- SQL
- C/C++
- Java™
- OLE
- COBOL

Note: For SQL procedures, you need to install and configure a supported C compiler on your database server.

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Stored Procedure Parameter Modes" on page 35
- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Setting Up the Application Development Environment" in the *Application Development Guide: Building and Running Applications*
- "Invoking Stored Procedures" on page 146

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "DB2 Supported Software for Building and Running Applications" in the *Application Development Guide: Building and Running Applications*

Routines: Scalar User-Defined Functions

Scalar UDFs enable you to extend and customize SQL. They can be invoked in the same manner as built-in DB2® functions (for example, LENGTH). That is, they can be referenced in SQL statements wherever an expression is valid. Scalar UDFs accept zero or more typed values as input arguments and return a single value.

Benefits

- Invoked as part of another SQL statement.
- Results can be processed directly by the invoking SQL statement.
- State can be maintained between invocations within a single SQL statement by using a scratchpad.

Limitations

- Cannot do transaction management.
- Cannot return result sets.
- Limited to single result at a time.
- Not designed for single invocations.

Common uses

- Perform logic inside an SQL statement that SQL cannot natively perform.
- Encapsulate scalar queries. For example, given a postal code, search a table for the city where the postal code is found.

Supported languages

- SQL
- C/C++
- Java™
- OLE

Notes:

1. There is a limited capability for creating column UDFs. Also known as aggregating functions, these receive a set of like values (a column of data) and return a single answer. A user-defined column function can only be created if it is sourced upon a built-in column function. 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)`.
2. You can also create UDFs that return a row. These are known as row UDFs and can only be used as a transform function for structured types. The output of a row UDF is a single row.

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Scratchpads for UDFs and Methods" on page 49

Related tasks:

- "Invoking UDFs" on page 148

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

Routines: Table User-Defined Functions

Like scalar UDFs, table UDFs enable you to extend and customize SQL, but for the purpose of generating tables. Table UDFs can only be invoked in the FROM clause of an SQL statement. Table UDFs accept zero or more typed values as input arguments and return a table.

Benefits

- Invoked as part of an SQL statement.
- Results can be directly processed by the invoking SQL statement.
- State can be maintained between invocations within a single SQL statement by using a scratchpad.
- Provides a set of data for processing.

Limitations

- Cannot do transaction management.

- Cannot return result sets.
- Not designed for single invocations.
- Can only be used in FROM clause.

Common uses

- Provide a tabular interface to non-relational data. For example, read a spreadsheet and produce a table, which could then be inserted into a DB2® table.
- Encapsulate a query. For example, given the name of a city, return all the postal codes that are valid for the city.

Supported languages

- SQL
- C/C++
- Java™
- OLE
- OLE DB

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Scratchpads for UDFs and Methods" on page 49
- "Table Function Processing Model" on page 55

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

Routines: Methods

Methods enable you to define behaviors for structured types. They are like scalar UDFs, but can only be defined for structured types. Methods share all the features of scalar UDFs, in addition to the following features:

Benefits

- Strongly associated with the structured type.
- Can be sensitive to the dynamic type of the subject type.

Limitations

- Can only return a scalar value.
- Can only be used with structured types.
- Cannot be invoked against typed tables.

Common uses

- Providing operations on structured types.
- Encapsulating the structured type.

Supported languages

- SQL
- C/C++
- Java™
- OLE

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Scratchpads for UDFs and Methods" on page 49

Related tasks:

- "Defining Behavior for Structured Types" on page 206

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Chapter 2. Developing Routines

Supported Routine Programming Languages

In general, routines are used to improve overall performance of the database management system by enabling application functionality to be performed on the database server. The amount of gain realized by these efforts is limited, to some degree, by the language chosen to write a routine.

Some of the issues you should consider before implementing routines in a certain language are:

- The available skills for developing a routine in a particular language and environment.
- The reliability and safety of a language's implemented code.
- The scalability of routines written in a particular language.

To help assess the preceding criteria, here are some characteristics of various supported languages:

SQL

- SQL routines are faster than Java™ routines, and roughly equivalent in performance to trusted C/C++ routines.

- SQL routines are written completely in SQL, making them quick to implement.
- SQL routines are considered ’safe’ by DB2® as they consist entirely of SQL statements. Because of this, SQL routines always run directly in the database engine, giving them good performance, and scalability.

C/C++

- Both C/C++ embedded SQL and DB2 CLI routines are faster than Java routines. They are roughly equivalent in performance to SQL routines when run in NOT FENCED mode.
- C/C++ routines are prone to error. It is recommended that you register C/C++ routines as FENCED NOT THREADSAFE, because routines in these languages are the most likely to damage the engine through memory corruption. Running in FENCED NOT THREADSAFE mode, while safer, incurs performance overhead.

  For information on assessing and mitigating the risks of registering C/C++ routines as NOT FENCED or FENCED THREADSAFE, see the topic, ″Security Considerations for Developing Routines″.
- By default, C/C++ routines run in FENCED NOT THREADSAFE mode to isolate them from damaging the execution of other routines. Because of this, you will have one `db2fmp` process per concurrently executing C/C++ routine on the database server. This can result in scalability problems on some systems.

Java

- Java routines are slower than C/C++ or SQL routines.
- Java routines are safer than C/C++ routines because control of dangerous operations is handled by the JVM. Because of this, reliability is increased, as it is difficult for a Java routine to damage another routine running in the same process.

  Note: To avoid potentially dangerous operations, JNI calls from Java routines are not permitted. If you need to invoke C/C++ code from a Java routine, you can do so by invoking a separately cataloged C/C++ routine.
- When run in FENCED THREADSAFE mode (the default), Java routines scale well. All FENCED Java routines will share a few JVMs (more than one JVM may be in use on the system if the Java heap of a particular db2fmp process is approaching exhaustion).
- Trusted Java routines scale well on Windows, where they share a single Java Virtual Machine (JVM), due to DB2’s multi-threaded engine. But they do not scale well on UNIX, where each DB2 agent running a Java routine must have its own dedicated JVM.

OLE

- OLE routines can be implemented in Visual C++, Visual Basic and other languages supported by OLE.
- The speed of OLE automated routines depends on the language used to implement them. In general, they are slower than non-OLE C/C++ routines.
- OLE routines can only run in FENCED NOT THREADSAFE mode. This minimizes the chance of engine corruption. This also means that OLE automated routines do not scale well.

OLE DB

- OLE DB can only be used to develop table functions.
- OLE DB table functions connect to a external OLE DB data source.
- Depending on the OLE DB provider, OLE DB table functions are generally faster than Java table functions, but slower than C/C++ or SQL-bodied table functions. However, some predicates from the query where the function is invoked may be evaluated at the OLE DB provider, therefore reducing the number of rows that DB2 has to process. This frequently results in improved performance.
- OLE DB routines can only run in FENCED NOT THREADSAFE mode. This minimizes the chance of engine corruption. This also means that OLE automated routines do not scale well.

Related concepts:

- "Performance Considerations for Developing Routines" on page 16
- "Security Considerations for Routines" on page 20
- "C/C++ Routines" on page 97
- "Java Routines" on page 118

Related tasks:

- "Building JDBC Routines" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ Routines" in the *Application Development Guide: Building and Running Applications*
- "Creating SQL Procedures" in the *Application Development Guide: Building and Running Applications*
- "Building CLI Routines on UNIX" in the *CLI Guide and Reference, Volume 1*
- "Building C Routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on AIX" in the *Application Development Guide: Building and Running Applications*

- “Building CLI Routines on Windows” in the *CLI Guide and Reference, Volume 1*
- “Building C/C++ Routines on Windows” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on HP-UX” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on HP-UX” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on Linux” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on Linux” in the *Application Development Guide: Building and Running Applications*
- “Building C Routines on Solaris” in the *Application Development Guide: Building and Running Applications*
- “Building C++ Routines on Solaris” in the *Application Development Guide: Building and Running Applications*

Best Practices for Developing Routines

The sections that follow feature recommended practices for developing secure routines that perform well.

Performance Considerations for Developing Routines

One of the most significant benefits of developing routines, instead of expanding client applications, is performance. Consider the following performance impacts when choosing an approach for routine implementation.

NOT FENCED mode

A NOT FENCED routine runs in the same process as the database manager. In general, running your routine as NOT FENCED results in better performance as compared with running it in FENCED mode, because FENCED routines run in a special DB2® process outside of the engine’s memory.

While you can expect improved routine performance when running routines in NOT FENCED mode, user code can accidentally or maliciously corrupt the database or damage the database control structures. You should only use NOT FENCED routines when you need to maximize the performance benefits, and if you deem the routine to be secure. (For information on assessing and mitigating the risks of registering C/C++ routines as NOT FENCED, see the topic, ″Security Considerations for Developing Routines″.) If the routine is not safe enough to run in the database manager’s process, use the FENCED clause when registering the routine.

If an abnormal termination occurs while you are running a NOT FENCED routine, the database manager will attempt an appropriate recovery if the routine is registered as NO SQL. However, for routines not defined as NO SQL, the database manager will fail.

NOT FENCED routines must be precompiled with the WCHARTYPE NOCONVERT option if the routine uses GRAPHIC or DBCLOB data.

FENCED THREADSAFE mode

FENCED THREADSAFE routines run in the same process as other routines. More specifically, non-Java routines share one process, while Java™ routines share another process, separate from routines written in other languages. This separation protects Java routines from the potentially more error prone routines written in other languages. Also, the process for Java routines contains a JVM, which incurs a high memory cost and is not used by other routine types. Multiple invocations of FENCED THREADSAFE routines share resources, and therefore incur less system overhead than FENCED NOT THREADSAFE routines, which each run in their own dedicated process.

If you feel your routine is safe enough to run in the same process as other routines, use the THREADSAFE clause when registering it. As with NOT FENCED routines, information on assessing and mitigating the risks of registering C/C++ routines as FENCED THREADSAFE is in the topic, ″Security Considerations for Developing Routines″.

If a FENCED THREADSAFE routine abends, only the thread running this routine is terminated. Other routines in the process continue running. However, the failure that caused this thread to abend may adversely affect other routine threads in the process, causing them to trap, hang, or have damaged data. After one thread abends, the damaged process is no longer used for new routine invocations. Once all the active users complete their jobs in this process, it is terminated.

When you register Java routines, they are deemed THREADSAFE unless you indicate otherwise. All other LANGUAGE types are NOT THREADSAFE by default. Routines using LANGUAGE OLE and OLE DB may not be specified as THREADSAFE.

NOT FENCED routines must be THREADSAFE. It is not possible to register a routine as NOT FENCED NOT THREADSAFE (SQLCODE -104).

Users on UNIX® can see their Java and C THREADSAFE processes by looking for `db2fmp (Java)` or `db2fmp (C)`.

FENCED NOT THREADSAFE mode

FENCED NOT THREADSAFE routines each run in their own dedicated process. If you are running numerous routines, this can have a detrimental effect on database system performance. If the routine is not safe enough to run in the same process as other routines, use the NOT THREADSAFE clause when registering the routine.

On UNIX, NOT THREADSAFE processes appear as `db2fmp (pid)` (where pid is the process id of the agent using the fenced mode process) or as `db2fmp (idle)` for a pooled NOT THREADSAFE db2fmp.

Java routines

For Java routines running on UNIX platforms, scalability may be an issue if NOT FENCED is specified. This is due to the nature of the DB2 UNIX process model, which is one process per agent. As a result, each invocation of a NOT FENCED Java routine will require its own JVM. This can result in poor scalability, because JVMs have a large memory footprint. Many invocations of NOT FENCED routines on a UNIX-based DB2 server will use a significant portion of system memory.

This is not the case for Java routines running on Windows® NT and Windows 2000, where each DB2 agent is represented by a thread in a process shared with other DB2 agent threads. This model is scalable as a single JVM is shared among all the DB2 agent threads in the process.

If you intend to run a Java routine with large memory requirements, it is recommended that you register it as FENCED NOT THREADSAFE. For FENCED THREADSAFE Java routine invocations, DB2 attempts to choose a threaded Java fenced mode process with a Java heap that is large enough to run the routine. Failure to isolate large heap consumers in their own process may result in out of Java heap errors in multithreaded Java db2fmp processes. If your Java routine does not fall into this category, FENCED routines will run better in threadsafe mode where they can share a small number of JVMs.

C/C++ routines

C or C++ routines are generally faster than Java routines, but are more prone to errors, memory corruption, and crashing. For these reasons, the ability to perform memory operations makes C or C++ routines risky candidates for THREADSAFE or NOT FENCED mode registration. These risks can be mitigated by adhering to programming practices for secure routines (see the topic, "Security Considerations for Developing Routines"), and thoroughly testing your routine.

SQL-bodied routines

SQL-bodied routines are also generally faster than Java routines, and usually share comparable performance with C routines. SQL routines always run in NOT FENCED mode, providing a further performance benefit over external routines.

Scratchpads

A scratchpad is a block of memory that can be assigned to UDFs and methods. The scratchpad only applies to the individual reference to the routine in an SQL statement. If there are multiple references to a routine in a statement, each reference has its own scratchpad. A scratchpad enables a UDF or method to save its state from one invocation to the next.

For UDFs and methods with complex initializations, you can use scratchpads to store any values required in the first invocation for use in all future invocations. Other UDFs and methods may require their values to be saved from invocation to invocation.

Use VARCHAR parameters instead of CHAR parameters

You can improve the performance of your routines 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 routine, 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 routine.

In comparison, passing the string ″A SHORT STRING″ as a VARCHAR(200) parameter to a routine results in DB2 simply passing the 14 character string and a null terminator across the network.

Related concepts:

- “WCHARTYPE Precompiler Option in C and C++” in the *Application Development Guide: Programming Client Applications*
- “WCHARTYPE CONVERT Precompile Option” in the *Application Development Guide: Building and Running Applications*
- “Security Considerations for Routines” on page 20
- “C/C++ Routines” on page 97
- “Java Routines” on page 118
- “Restrictions for Routines” on page 24
- “Library and Class Management Considerations for Developing Routines” on page 23

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Security Considerations for Routines

Developing and deploying routines provides you with an opportunity to greatly improve the performance and effectiveness of your database applications. There can, however, be security risks if the deployment of routines is not managed correctly by the database administrator. The following sections describe security risks and means by which you can mitigate these risks. The security risks are followed by a section on how to safely deploy routines whose security is unknown.

Security Risks:

NOT FENCED routines can access database manager resources

NOT FENCED routines run in the same process as the database manager. Because of their close proximity to the database engine, NOT FENCED routines can accidentally or maliciously corrupt the database manager's shared memory, or damage the database control structures. Either form of damage will cause the database manager to fail. NOT FENCED routines can also corrupt databases and their tables.

To ensure the integrity of the database manager and its databases, you must thoroughly screen routines you intend to register as NOT FENCED. These routines must be fully tested, debugged, and exhibit no unexpected side-effects. In the examination of the routine, pay close attention to memory management and the use of static variables. This is where the greatest potential for corruption lies, particularly in languages other than Java.

In order to register a NOT FENCED routine, the CREATE_NOT_FENCED_ROUTINE authority is required. When granting the CREATE_NOT_FENCED_ROUTINE authority, be aware that the recipient can potentially gain unrestricted access to the database manager and all its resources.

FENCED THREADSAFE routines can access memory in other FENCED THREADSAFE routines

FENCED THREADSAFE routines run as threads inside a shared process. Each of these routines are able to read the memory used by other routine threads in the same process. Therefore, it is possible for one threaded routine to collect sensitive data from other routines in the threaded process. Another risk inherent in the sharing of a single process, is that one routine thread with flawed memory management can corrupt other routine threads, or cause the entire threaded process to crash.

To ensure the integrity of other FENCED THREADSAFE routines, you must thoroughly screen routines you intend to register as FENCED THREADSAFE. These routines must be fully tested, debugged, and exhibit no unexpected side-effects. In the examination of the routine, pay close attention to memory management and the use of static variables. This is where the greatest potential for corruption lies, particularly in languages other than Java.

In order to register a FENCED THREADSAFE routine, the CREATE_EXTERNAL_ROUTINE authority is required. When granting the CREATE_EXTERNAL_ROUTINE authority, be aware that the recipient can potentially monitor or corrupt the memory of other FENCED THREADSAFE routines.

Write access to the database server by the owner of fenced processes can result in database manager corruption

The user ID under which fenced processes run is defined by the **db2icrt** (create instance) or **db2iupdt** (update instance) system commands. This user ID must not have write access to the directory where routine libraries and classes are stored (in UNIX® environments, sqllib/function; in Windows® environments, sqllib\function). This user ID must also not have write access to any database, operating system, or otherwise critical files and directories on the database server.

If the owner of fenced processes does have write access to various critical resources on the database server, the potential for system corruption exists. For example, a database administrator registers a routine received from an unknown source as FENCED NOT THREADSAFE, thinking that any potential harm can be averted by isolating the routine in its own process. However, the user ID that owns fenced processes has write access to the sqllib/function directory. Users invoke this routine, and unbeknownst to them, it overwrites a library in sqllib/function with an alternate version of a routine body that is registered as NOT FENCED. This second routine has unrestricted access to the entire database manager, and can thereby distribute sensitive information from database tables, corrupt the databases, collect authentication information, or crash the database manager.

Ensure the user ID that owns fenced processes does not have write access to critical files or directories on the database server (especially sqllib/function and the database data directories).

Vulnerability of routine libraries and classes
If access to the directory where routine libraries and classes are stored is not controlled, routine libraries and classes can be deleted or overwritten. As discussed in the previous item, the replacement of a NOT FENCED routine body with a malicious (or poorly coded) routine can severely compromise the stability, integrity, and privacy of the database server and its resources.

To protect the integrity of routines, you must manage access to the directory containing the routine libraries and classes. Ensure that the fewest possible number of users can access this directory and its files. When assigning write access to this directory, be aware that this privilege can provide the owner of the user ID unrestricted access to the database manager and all its resources.

Deploying potentially insecure routines:

If you happen to acquire a routine from an unknown source, be sure you know exactly what it does before you build, register, and invoke it. It is recommend that you register it as FENCED and NOT THREADSAFE unless you have tested it thoroughly, and it exhibits no unexpected side-effects.

If you need to deploy a routine that does not meet the criteria for secure routines, register the routine as FENCED and NOT THREADSAFE. To ensure that database integrity is maintained, FENCED and NOT THREADSAFE routines:

- Run in a separate DB2® process, shared with no other routines. If they abnormally terminate, the database manager will be unaffected.
- Use memory that is distinct from memory used by the database. An inadvertent mistake in a value assignment will not affect the database manager.

Related concepts:
- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Performance Considerations for Developing Routines" on page 16
- "Restrictions for Routines" on page 24
- "Library and Class Management Considerations for Developing Routines" on page 23

Related reference:
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "GRANT (Routine Privileges) statement" in the *SQL Reference, Volume 2*
- "REVOKE (Routine Privileges) statement" in the *SQL Reference, Volume 2*

Library and Class Management Considerations for Developing Routines

The bodies of external routines reside in libraries and classes stored on the database server. These files are not backed up or protected in any way by DB2. The CREATE statements that register routines merely add routine definition information to the database catalogs. The routine library or class exists only in the location it was installed, and it could be deleted or altered while the catalog entry referring to it remains unchanged.

To preserve the integrity of the invoking clients and routines that depend on the routine, you must prevent the routine body from being inadvertently or intentionally deleted or replaced. This can be done by managing access to the directory containing the routine and by protecting the routine body itself.

If you need to change the body of a routine, do not recompile and relink the routine to the same file name (for example, sqllib/function/foo.a) the current routine is using while the database manager is running. Operating system level caching can cause such an operation to fail. If it is necessary to change the body of a routine without stopping and restarting DB2, complete the following steps:

1. Create the new body for the routine with a different library or class name.
2. Use the ALTER statement to change the routine's EXTERNAL NAME to reference the updated routine body.

Once the ALTER updates the routine's catalog entries, all subsequent invocations of the updated routine will point to the new routine body.

For updating Java™ routines that are built into JAR files, you must 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. DB2 refreshes the classes when a COMMIT or ROLLBACK occurs. The CALL SQLJ.REFRESH_CLASSES() statement only applies to FENCED routines. To update NOT FENCED routines, you must either restart the database manager and replace the class, or use the steps described above to create a new class and use the ALTER statement to reference it.

The DB2 library manager dynamically adjusts its library caching according to your workload. For optimal performance consider the following:

- Keep the number of routines in your libraries as small as possible. If you are including multiple routines in the same library, ensure that you group them based on if they are invoked in the same time frame. Consider a scenario where in a number of applications a call to the stored procedure, ProcA is followed by a call to the stored procedure, ProcB. This is a case where it may be appropriate to include ProcA and ProcB in the same library. With a library caching scheme, it is better to have numerous smaller libraries than few large libraries.

- The load cost for a library in the THREADSAFE C process is paid only once for libraries that are consistently in use by THREADSAFE C routines. After the routine's first invocation, all subsequent invocations, from any thread in the process, do not need to load the routine's library.

Note: The bodies of SQL-bodied routines are part of the database, and as such, will be backed up with other database objects. However, like external routines, their bodies are prone to being altered, and therefore require the same protection.

Related concepts:

- "Performance Considerations for Developing Routines" on page 16
- "Security Considerations for Routines" on page 20
- "Restrictions for Routines" on page 24

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*
- "ALTER FUNCTION statement" in the *SQL Reference, Volume 2*
- "ALTER METHOD statement" in the *SQL Reference, Volume 2*
- "ALTER PROCEDURE statement" in the *SQL Reference, Volume 2*

Restrictions for Routines

The following are restrictions for developing routines.

- In pre-Version 8 editions of DB2, CALL was not a compiled statement and data type matching was not enforced. The data types you register a routine with must match the data types used in the routines. See the tables with SQL type mappings to Java, C, OLE automation, and OLE DB data types.
- UDFs cannot return result sets. All cursors opened by a UDF with SQL must be closed by the time the final call is completed.
- Routines should not create new threads.
- You cannot issue any connection level APIs from UDFs or methods.
- Input to, and output from the screen and keyboard is not possible from routines. Hence, you should not use the standard I/O streams; for example, calls to `System.out.println()` in Java, `printf()` in C/C++, or `display` in

COBOL. In the process model of DB2, routines run in the background and cannot write to the screen. However, routines can write to a file.

For FENCED routines that run on UNIX, the target directory where the file is to be created, or the file itself, must have the appropriate permissions such that the owner of the `sqllib/adm/.fencedid` file can create it or write to it. For NOT FENCED routines, the instance owner must have create and write permissions for the directory in which the file is opened.

Note: DB2® does not attempt to synchronize any external input or output performed by a routine 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.

- You cannot execute any connection-related statements or commands in routines, including:
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- In general, DB2 does not restrict the use of operating system functions. However, there are a few exceptions:
  1. **It is imperative that no routine install its own signal handlers.** Failure to adhere to this restriction can result in unexpected failures, database abends, or other problems. Installing signal handlers may also interfere with operation of the JVM for Java™ routines.
  2. 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 routines containing system calls.
- Routines must not contain commands that would terminate the current process. A routine must always return control to DB2 without terminating the current process.
- When returning result sets from nested stored procedures, you can open a cursor with the same name on multiple nesting levels. However,

pre-version 8 applications will only be able to access the first result set that was opened. This restriction does not apply to cursors that are opened with a different package level.

- Do not change the bodies of routines while the database is active. If it is necessary to change the body of a routine without stopping and restarting DB2, create the new body for the routine with a different library name. The ALTER statement can then be used to change the routine's EXTERNAL NAME to reference the new body.
- The values of all environment variables with names beginning with 'DB2' are captured at the time the database manager is started with `db2start`, and are available in all routines, whether they are FENCED or NOT FENCED. The only exception is the `DB2CKPTR` environment variable. Other environment variables are accessible from NOT FENCED routines, but not from the FENCED routine process (for example, `LIBPATH`). Note that the environment variables are *captured*. Any changes to the environment variables after `db2start` is issued are not available to the routines.
- When using protected resources (resources that allow only one process access at a time inside routines), you should try to avoid deadlocks between routines. If two or more routines deadlock, DB2 will not be able to detect or resolve the condition, resulting in hung routine processes.
- If you allocate dynamic memory in a routine, it should be freed before returning to DB2. Failure to do so results in a memory leak, and the continual growth of DB2 processes, which could eventually lead to out-of-memory conditions.

  For UDFs and methods, the scratchpad facility can be used to anchor dynamic memory needed across multiple invocations. If you use a scratchpad in this manner, specify the FINAL CALL attribute in the CREATE statement for the UDF or method so that it can free the allocated memory at end-of-statement processing.
- Do not allocate storage for any parameters in your routine on the database server. The database manager automatically allocates storage based upon the parameter declaration in the CREATE statement. Do not alter any storage pointers for parameters in the routine. Attempting to change a pointer with a locally created storage pointer may result in memory leaks, data corruption, or abends.
- Do not use static or global data in routines. DB2 cannot guarantee that the memory used by static or global variables will be untouched between routine invocations. For UDFs and methods, you can use scratchpads to store values for use between invocations.
- All SQL argument values are buffered. This means that a copy of the value is made and presented to the routine. If there are changes made to the input parameters of a routine, these changes will have no effect on SQL values or processing. However, if a routine writes more data to an input or

output parameter than is specified by the CREATE statement, memory corruption has occurred, and the routine can abend.

Related concepts:

- "Performance Considerations for Developing Routines" on page 16
- "Security Considerations for Routines" on page 20
- "SQL Data Type Handling in C/C++ Routines" on page 106

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in C and C++" in the *Application Development Guide: Programming Client Applications*
- "Data Type Mappings between DB2 and OLE DB" in the *Application Development Guide: Programming Client Applications*
- "ALTER FUNCTION statement" in the *SQL Reference, Volume 2*
- "ALTER METHOD statement" in the *SQL Reference, Volume 2*
- "ALTER PROCEDURE statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in OLE DB" on page 143
- "Supported SQL Data Types in OLE Automation" on page 133

Registering Routines

Registering a routine is the act of coupling a custom-built application library with the database. Until the routine is registered, it cannot be invoked as a routine.

For the routine to work properly, it is vital that you register it with the applicable clauses. Choices you made while writing the routine need to be reflected in its registration. For example, there needs to be an exact mapping between the parameters passed from a client application to a routine. To simplify matters, many of the clauses for registering the different types of routines are common.

Prerequisites:

For the list of privileges required to register routines, see the following statements:

- CREATE FUNCTION

- CREATE METHOD
- CREATE TYPE
- CREATE PROCEDURE

Procedure:

To register a routine, issue the CREATE statement with the applicable clauses that correspond to the type of routine you are working with. The statements are as follows: CREATE FUNCTION, CREATE METHOD, CREATE TYPE, and CREATE PROCEDURE.

For the registration of methods, issuing the CREATE TYPE statement is the first step, and issuing the CREATE METHOD statement is the second step. The CREATE METHOD statement only addresses attributes that relate to a method's signature.

Once you have registered your routine, you can invoke it from a client application or a calling routine.

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "Parameter Styles for External Routines" on page 71
- "Performance Considerations for Developing Routines" on page 16
- "Security Considerations for Routines" on page 20

Related tasks:

- "Writing Routines" on page 29

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Related samples:

- "spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc "
- "spserver.db2 -- To create a set of SQL procedures "
- "UDFsCreate.db2 -- How to catalog the UDFs contained in UDFsqlsv.java "

Writing Routines

Each of the three types of routines (stored procedures, UDFs, and methods) share a great deal in common with regards to how they are written. For instance, they employ some of the same parameter styles, they support the use of SQL through various client interfaces (embedded SQL, CLI, and JDBC), and they can invoke other routines. To this end, the following steps represent a single approach for writing routines.

There are some tasks that are not common in the writing of all types of routines. For example, result sets are specific to stored procedures, and scratchpads are specific to UDFs and methods. When you come across a step not applicable to the type of routine you are developing, go to the step that follows it.

Prerequisites:

Before writing a routine, you must decide the following:
- The type of routine you need. (See Routines: Stored Procedures.)
- The programming language you will use to write it. (See Supported Routine Programming Languages.)
- Which interface to use if you require SQL statements in your routine. (See When to Use DB2 CLI or Embedded SQL.)

See also the topics on Security, Library and Class Management, and Performance considerations.

Procedure:

To create a routine body, you must:
1. *Applicable only to external routines.* Accept input parameters from the invoking application or routine and declare output parameters. How a routine accepts parameters is dependent on the parameter style you will register the routine with. Each parameter style defines the set of parameters that are passed to the routine body and the order that the parameters are passed.

   For example, the following is a signature of a UDF body written in C (using sqludf.h) for PARAMETER STYLE SQL:
   ```
   SQL_API_RC SQL_API_FN product ( SQLUDF_DOUBLE *in1,
                                   SQLUDF_DOUBLE *in2,
                                   SQLUDF_DOUBLE *outProduct,
                                   SQLUDF_NULLIND *in1NullInd,
                                   SQLUDF_NULLIND *in2NullInd,
                                   SQLUDF_NULLIND *productNullInd,
                                   SQLUDF_TRAIL_ARGS )
   ```

2. Add the logic that the routine is to perform. Some tools at your disposal for writing the body of the routine are as follows:
   - Calling other routines (nesting), or calling the current routine (recursion).
   - In routines that are defined to have SQL (CONTAINS SQL, READS SQL, or MODIFIES SQL), the routine may issue SQL statements. The types of statements that can be invoked is controlled by how routines are registered.
   - In external UDFs and methods, use scratchpads to save state from one call to the next.
   - In SQL procedures, use condition handlers to determine the SQL procedure’s behavior when a specified condition occurs. You can define conditions based on SQLSTATEs.
3. *Applicable only to stored procedures.* Return one or more result sets. In addition to individual parameters that are exchanged with the calling application, stored procedures have the capability to return multiple result sets. Only SQL routines and CLI, ODBC, JDBC, and SQLj routines and clients can accept result sets.

In addition to writing your routine, you also need to register it before you can invoke it. This is done with the CREATE statement that matches the type of routine you are developing. In general, the order in which you write and register your routine does not matter. However, the registration of a routine must precede its being built if it issues SQL that references itself. In this case, for a bind to be successful, the routine’s registration must have already occurred.

Related concepts:
- “When to Use DB2 CLI or Embedded SQL” in the *Application Development Guide: Programming Client Applications*
- “Parameter Styles for External Routines” on page 71
- “Performance Considerations for Developing Routines” on page 16
- “Security Considerations for Routines” on page 20
- “C/C++ Routines” on page 97
- “Java Routines” on page 118
- “Restrictions for Routines” on page 24
- “Library and Class Management Considerations for Developing Routines” on page 23
- “OLE Automation Routine Design” on page 130
- “OLE DB User-Defined Table Functions” on page 138
- “Supported Routine Programming Languages” on page 13

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Related samples:

- "spserver.c -- Definition of various types of stored procedures (CLI)"
- "spserver.db2 -- To create a set of SQL procedures "
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)"
- "SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)"

Debugging Routines

Before deploying routines on a production server you must thoroughly test and debug them on a test server. This is especially important for routines that need to be registered as NOT FENCED because they have unrestricted access to the database manager's memory, its databases, and database control structures. FENCED THREADSAFE routines also demand close attention because they share memory with other routines.

Procedure:

Checklist of common routine problems

To ensure that a routine executes properly, check that:

- The routine is registered properly. The parameters provided in the CREATE statement must match the arguments handled by the routine body. With this in mind, check the following specific items:
  - The data types of the arguments used by the routine body are appropriate for the parameter types defined in the CREATE statement.
  - The routine does not write more bytes to an output variable than were defined for the corresponding result in the CREATE statement.
  - The routine arguments for SCRATCHPAD, FINAL CALL, DBINFO are present if the routine was registered with corresponding CREATE options.

- For external routines, the value for the EXTERNAL NAME clause in the CREATE statement must match the routine library and entry point (case sensitivity varies by platform).
- For C++ routines, the C++ compiler applies type decoration to the entry point name. Either the type decorated name needs to be specified in the EXTERNAL NAME clause, or the entry point should be defined as `extern "C"` in the user code.
- The routine name specified during invocation must match the registered name (defined in the CREATE statement) of the routine. By default, routine identifiers are folded to uppercase. This does not apply to delimited identifiers, which are not folded to uppercase, and are therefore case sensitive.

  The routine must be placed in the directory path specified in the CREATE statement, or if no path is given, where DB2 looks for it by default. For UDFs, methods, and fenced stored procedures, this is: `sqllib/function` (UNIX) or `sqllib\function` (Windows). For unfenced stored procedures, this is: `sqllib/function/unfenced` (UNIX) or `sqllib\function\unfenced` (Windows).

- The routine is built using the correct calling sequence, precompile (if embedded SQL), compile, and link options.
- The application is bound to the database, except if it is written using DB2 CLI, ODBC, or JDBC. The routine must also be bound if it contains SQL and does not use any of these interfaces.
- The routine accurately returns any error information to the client application.
- All applicable call types are accounted for if the routine was defined with FINAL CALL.
- The system resources used by routines are returned.

Routine debugging techniques

To debug a routine, use the following techniques:

- The Development Center provides extensive debugging tools for SQL-bodied and Java stored procedures.
- It is not possible to write diagnostic data to screen from a routine. If you intend to write diagnostic data to a file, ensure that you write to a globally accessible directory such as \tmp. Do not write to directories used by database managers or databases.

  For stored procedures, a safe alternative is to write diagnostic data to an SQL table. The stored procedure you are testing must be registered with the MODIFIES SQL DATA clause in order to be able to write to an SQL table. If you need an existing stored procedure to write data (or no longer write data) to an SQL table, you must drop and re-register the stored procedure with (or without) the

MODIFIES SQL DATA clause. Before dropping and re-registering the stored procedure, be aware of its dependencies.
- You can debug your routine locally by writing a simple application that invokes the routine entry point directly. Consult your compiler documentation for information on using the supplied debugger.

Related concepts:
- "Security Considerations for Routines" on page 20

Related tasks:
- "Debugging Stored Procedures in Java" on page 125
- "Displaying Error Messages for SQL Procedures" on page 62
- "Debugging : Development Center help" in the *Help: Development Center*

Related reference:
- "Identifiers" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in OLE DB" on page 143
- "Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL" on page 74
- "Supported SQL Data Types in OLE Automation" on page 133
- "Supported SQL Data Types in C/C++" on page 103

Conflicts When Reading and Writing Tables From Routines

To preserve the integrity of the database, it is necessary to avoid conflicts when reading and writing to tables. For example, suppose an application is updating the EMPLOYEE table, and the statement calls a routine. Suppose that the routine tries to read the EMPLOYEE table and encounters the row being updated. The routine sees the row while it is in an indeterminate state - perhaps some columns of the row have been updated while other have not. If the routine acts on this partially updated row, it can take incorrect actions. To avoid this sort of problem, DB2® does not allow operations that conflict on any table.

To describe how DB2 avoids conflicts when reading and writing tables from routines, the following two terms are needed:

top-level statement
: A top-level statement is any SQL statement issued from an application, or from a stored procedure that was invoked as a top-level statement.

table access context
: A table access context refers to the scope where conflicting operations on a table are allowed. A table access context is created whenever:

  - A top-level statement issues an SQL statement.
  - A UDF or method is invoked.

  For example, when an application calls a stored procedure, the CALL is a top-level statement and therefore gets a table access context. If the stored procedure does an UPDATE, the UPDATE is also a top-level statement (since the stored procedure was invoked as a top-level statement) and therefore gets a table access context. If the UPDATE invokes a UDF, the UDF gets a separate table access context and SQL statements inside the UDF are not top-level statements.

Once a table has been read or written, it is protected from conflicts within the top-level statement that made the access. The table can be read or written from a different top-level statement or from a routine invoked from a different top-level statement.

The following rules are applied:

1. Within a table access context, a given table may be both read and written without causing a conflict.
2. If a table is being read within a table access context then other contexts may also read the table. If any other context attempts to write to the table, however, a conflict occurs.
3. If a table is being written within a table access context, then no other context may read or write to the table without causing a conflict.

If a conflict occurs, an error (SQLCODE -746, SQLSTATE 57053) is returned to the statement that caused the conflict.

The following is an example of table read and write conflicts:

Suppose an application issues the statement:

```
UPDATE t1 SET c1 = udf1(c2)
```

UDF1 contains the statements:

```
DECLARE cur1 CURSOR FOR SELECT c1, c2 FROM t1
OPEN cur1
```

This will result in a conflict because rule 3 is violated. This form of conflict can only be resolved by redesigning the application or UDF.

The following does not result in a conflict:

Suppose an application issues the statements:

```
DECLARE cur2 CURSOR FOR SELECT udf2(c1) FROM t2
OPEN cur2
FETCH cur2 INTO :hv
UPDATE t2 SET c2 = 5
```

UDF2 contains the statements:

```
DECLARE cur3 CURSOR FOR SELECT c1, c2 FROM t2
OPEN cur3
FETCH cur3 INTO :hv
```

With the cursor, UDF2 is allowed to read table T2 since two table access contexts can read the same table. The application is allowed to update T2 even though UDF2 is reading the table because UDF2 was invoked in a different application level statement than the update.

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3
- "SQL in External Routines" on page 89

Stored Procedure Features

Stored Procedures have special capabilities for exchanging data with invoking applications and routines. The sections that follow describe stored procedure parameter modes, the capability of stored procedures to return result sets, and the option of accepting parameters in the style of a main routine or a subroutine.

Stored Procedure Parameter Modes

Client applications and calling routines exchange information with stored procedures through parameters and result sets. The parameters for routines are defined as having specific data types. Unlike other routines, the parameters for stored procedures are also defined by the direction the data is traveling (the parameter mode).

There are three types of parameters for stored procedures:

- IN parameters: data passed to the stored procedure.
- OUT parameters: data returned by the stored procedure.

- INOUT parameters: data passed to the stored procedure that is, during stored procedure execution, replaced by data to be returned from the stored procedure.

The mode of parameters and their data types are defined when a stored procedure is registered with the CREATE PROCEDURE statement.

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Registering Routines" on page 27

Related reference:

- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*

Stored Procedure Result Sets

The following sections describe the ability of stored procedures to return result sets, and how to return and receive them using various interfaces.

Stored Procedure Result Sets

In addition to exchanging parameters, stored procedures can pass information to invokers by returning result sets. Result sets can be accepted by SQL-bodied routines, and routines and applications programmed in the following interfaces:

- CLI
- JDBC
- SQLj
- ODBC

Stored procedures pass result sets to their invokers through cursors. The stored procedure body must contain a cursor for every result set you need to return. While you can fetch rows from a result set cursor within the stored procedure, only unfetched rows are passed to the invoker as the result set. When exiting a stored procedure, leave the cursors that correspond to the result sets open. Multiple result sets are returned in the order in which you open their cursors.

When declaring a cursor for a result set, it is strongly recommended that you specify the destination in the WITH RETURN TO clause of the DECLARE CURSOR statement (for SQL procedures, this is mandatory). To return the result set to the invoker, whether the invoker is an application or a routine, specify WITH RETURN TO CALLER. To return the result set directly to the application, bypassing any intermediate nested routines, specify WITH

RETURN TO CLIENT. In external routines, cursors are defined as WITH RETURN TO CALLER by default, unless they are explicitly defined as WITH RETURN TO CLIENT.

When registering a stored procedure with the CREATE PROCEDURE statement, indicate the number of result sets that it returns with the DYNAMIC RESULT SETS clause. This value is stored in the RESULT_SETS column in the SYSCAT.ROUTINES view. If the number of result sets returned from a stored procedure is different than the number specified in the CREATE PROCEDURE statement, a warning is issued (SQLCODE +464, SQLSTATE 0100E). For PARAMETER STYLE JAVA stored procedures, the number of result sets in the CREATE PROCEDURE statement must match the number of ResultSet[] parameters in the Java™ method signature.

The invoker can DESCRIBE the received result sets. Note that if the same cursor is opened on multiple nesting levels, applications running on DB2® UDB Version 7 clients can only DESCRIBE the first result set that is opened.

Result sets must be processed in a serial fashion by the invoker (if the invoker is not an SQL-bodied routine). 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.

To receive result sets in SQL-bodied routines, you must DECLARE and ASSOCIATE result set locators to the stored procedure you expect will return result sets. You must then ALLOCATE each cursor you expect will be returned to a result set locator. Once this is done, you can fetch rows from the result sets.

Note: A COMMIT issued from within the stored procedure or from the application will close any result sets that are not for WITH HOLD cursors. A ROLLBACK issued from the application or from the stored procedure will close all result set cursors. After a COMMIT or a ROLLBACK is made from within a stored procedure, cursors may be opened and returned as result sets.

Related concepts:

- "Routines: Stored Procedures" on page 7
- "Cursors in CLI Applications" in the *CLI Guide and Reference, Volume 1*
- "Result Set Terminology in CLI Applications" in the *CLI Guide and Reference, Volume 1*
- "Result Set Retrieval into Arrays in CLI Applications" in the *CLI Guide and Reference, Volume 1*

Related tasks:

- "Declaring and Using Cursors in Static SQL Programs" in the *Application Development Guide: Programming Client Applications*
- "Declaring and Using Cursors in Dynamic SQL Programs" in the *Application Development Guide: Programming Client Applications*
- "Returning Result Sets From SQL and Embedded SQL Stored Procedures" on page 38
- "Receiving Stored Procedure Result Sets in SQL-bodied Routines" on page 42
- "Receiving Stored Procedure Result Sets in JDBC Applications and Routines" on page 45
- "Returning Result Sets From JDBC Stored Procedures" on page 41
- "Receiving Stored Procedure Result Sets in SQLj Applications and Routines" on page 44
- "Returning Result Sets From SQLj Stored Procedures" on page 40

Related reference:

- "COMMIT statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "DESCRIBE statement" in the *SQL Reference, Volume 2*
- "PREPARE statement" in the *SQL Reference, Volume 2*
- "ROLLBACK statement" in the *SQL Reference, Volume 2*
- "SYSCAT.ROUTINES catalog view" in the *SQL Reference, Volume 1*

Returning Result Sets From SQL and Embedded SQL Stored Procedures

You can develop stored procedures that return result sets to the invoking routine or application. In SQL and embedded SQL stored procedures, the returning of result sets is handled with the DECLARE CURSOR statement.

Procedure:

To return a result set from an SQL or embedded SQL stored procedure:

1. Declare a cursor using the DECLARE CURSOR statement. The cursor declaration includes the SELECT statement that generates the set of rows that will compose the result set. In the cursor declaration it is strongly recommended that you specify the result set destination with the WITH RETURN TO clause (this is mandatory for SQL procedures).
   - To return a result set to the invoker of a stored procedure, whether the invoker is a client application or another routine, use the WITH RETURN TO CALLER clause.

In the following example, the SQL procedure “CALLER_SET” uses the WITH RETURN TO CALLER clause to return a result set to the invoker 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
```

- To return a result set from a stored procedure to the originating application, use the WITH RETURN TO CLIENT clause. When WITH RETURN TO CLIENT is specified on a result set, no nested stored procedures can access 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 invoked as a nested routine:

```
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
```

2. Open the cursor using the OPEN statement. After the cursor is opened in the stored procedure, you can FETCH rows from it. However, the result set that is returned to the application or calling routine will only contain unfetched rows.
3. Exit from the stored procedure without closing the cursor.

If you have not done so already, develop a client application or caller routine that will accept result sets from your stored procedure.

Related concepts:

- “Condition Handlers in SQL Procedures” on page 63
- “SQLCODE and SQLSTATE Variables in SQL Procedures” on page 68
- “Stored Procedure Result Sets” on page 36

Related tasks:

- “Creating SQL Procedures” in the *Application Development Guide: Building and Running Applications*
- “Calling Stored Procedures with the CALL Statement” in the *Application Development Guide: Building and Running Applications*
- “Calling SQL Procedures with Client Applications on UNIX” in the *Application Development Guide: Building and Running Applications*
- “Calling SQL Procedures with Client Applications on Windows” in the *Application Development Guide: Building and Running Applications*
- “Receiving Stored Procedure Result Sets in SQL-bodied Routines” on page 42
- “Receiving Stored Procedure Result Sets in JDBC Applications and Routines” on page 45
- “Receiving Stored Procedure Result Sets in SQLj Applications and Routines” on page 44

Related reference:

- “SQL Procedure Samples” in the *Application Development Guide: Building and Running Applications*

Related samples:

- “spserver.sqc -- A variety of types of stored procedures (C)”
- “spserver.sqC -- A variety of types of stored procedures (C++)”

Returning Result Sets From SQLj Stored Procedures

You can develop SQLj stored procedures that return result sets to the invoking routine or application. In SQLj stored procedures, the returning of result sets is handled with ResultSet objects.

Procedure:

To return a result set from an SQLj stored procedure:

1. Declare an iterator class to handle query data. For example:
   ```
   #sql iterator SpServerEmployees(String, String, double);
   ```
2. For each result set that is to be returned, include a parameter of type ResultSet[] in the stored procedure declaration. For example the following function signature accepts an array of ResultSet objects:
   ```
   public static void getHighSalaries(
     double inSalaryThreshold,        // double input
     int[] errorCode,                 // SQLCODE output
     ResultSet[] rs)                  // ResultSet output
   ```
3. Instantiate an iterator object. For example:
   ```
   SpServerEmployees c1;
   ```

4. Assign the SQL statement that will generate the result set to an iterator. In the following example, a host variable (called inSalaryThreshold -- see the function signature example above) is used in the query's WHERE clause:

   ```
   #sql c1 = {SELECT name, job, CAST(salary AS DOUBLE)
                FROM staff
                WHERE salary > :inSalaryThreshold
                ORDER BY salary};
   ```
5. Execute the statement and get the result set:

   ```
   rs[0] = c1.getResultSet();
   ```

If you have not done so already, develop a client application or caller routine that will accept result sets from your stored procedure.

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Receiving Stored Procedure Result Sets in SQL-bodied Routines" on page 42
- "Receiving Stored Procedure Result Sets in JDBC Applications and Routines" on page 45
- "Receiving Stored Procedure Result Sets in SQLj Applications and Routines" on page 44

Related samples:

- "SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)"

Returning Result Sets From JDBC Stored Procedures

You can develop JDBC stored procedures that return result sets to the invoking routine or application. In JDBC stored procedures, the returning of result sets is handled with ResultSet objects.

Procedure:

To return a result set from a JDBC stored procedure:

1. For each result set that is to be returned, include a parameter of type ResultSet[] in the stored procedure declaration. For example, the following function signature accepts an array of ResultSet objects:

   ```
   public static void getHighSalaries(
     double inSalaryThreshold,         // double input
     int[] errorCode,                  // SQLCODE output
     ResultSet[] rs)                   // ResultSet output
   ```
2. Open the invoker's database connection (using a Connection object):

```
Connection con =
    DriverManager.getConnection("jdbc:default:connection");
```

3. Prepare the SQL statement that will generate the result set (using a PreparedStatement object). In the following example, the prepare is followed by the assignment of an input variable (called inSalaryThreshold -- see the function signature example above) to the value of the parameter marker (a parameter marker is indicated with a ″?″) in the previous statement.

   ```
   String query =
       "SELECT name, job, CAST(salary AS DOUBLE) FROM staff " +
       "  WHERE salary > ? " +
       "  ORDER BY salary";

   PreparedStatement stmt = con.prepareStatement(query);
   stmt.setDouble(1, inSalaryThreshold);
   ```

4. Execute the statement:

   ```
   rs[0] = stmt.executeQuery();
   ```

5. End the stored procedure body.

If you have not done so already, develop a client application or caller routine that will accept result sets from your stored procedure.

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Receiving Stored Procedure Result Sets in SQL-bodied Routines" on page 42
- "Receiving Stored Procedure Result Sets in JDBC Applications and Routines" on page 45
- "Receiving Stored Procedure Result Sets in SQLj Applications and Routines" on page 44

Related samples:

- "SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)"

Receiving Stored Procedure Result Sets in SQL-bodied Routines

You can receive result sets from stored procedures you invoke from within an SQL-bodied routine.

Prerequisites:

You must know how many result sets the invoked stored procedure will return. For each result set that the invoking routine receives, a result set must be declared.

Procedure:

To accept stored procedure result sets from within an SQL-bodied routine:

1. DECLARE result set locators for each result set that the stored procedure will return. For example:
   ```
   DECLARE result1 RESULT_SET_LOCATOR VARYING;
   DECLARE result2 RESULT_SET_LOCATOR VARYING;
   DECLARE result3 RESULT_SET_LOCATOR VARYING;
   ```
2. Invoke the stored procedure. For example:
   ```
   CALL targetProcedure();
   ```
3. ASSOCIATE the result set locator variables (defined above) with the invoked stored procedure. For example:
   ```
   ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
      WITH PROCEDURE targetProcedure;
   ```
4. ALLOCATE the result set cursors passed from the invoked stored procedure to the result set locators. For example:
   ```
   ALLOCATE rsCur CURSOR FOR RESULT SET result1;
   ```
5. FETCH rows from the result sets. For example:
   ```
   FETCH rsCur INTO ...
   ```

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Returning Result Sets From SQL and Embedded SQL Stored Procedures" on page 38
- "Returning Result Sets From JDBC Stored Procedures" on page 41
- "Returning Result Sets From SQLj Stored Procedures" on page 40

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*
- "DECLARE CURSOR statement" in the *SQL Reference, Volume 2*
- "FETCH statement" in the *SQL Reference, Volume 2*
- "ALLOCATE CURSOR statement" in the *SQL Reference, Volume 2*
- "ASSOCIATE LOCATORS statement" in the *SQL Reference, Volume 2*

Receiving Stored Procedure Result Sets in SQLj Applications and Routines

You can receive result sets from stored procedures you invoke from an SQLj routine or application.

Procedure:

To accept stored procedure result sets from within an SQLj routine or application:

1. Open a database connection (using a Connection object):

   ```
   Connection con =
       DriverManager.getConnection("jdbc:db2:sample", userid, passwd);
   ```
2. Set the default context (using a DefaultContext object):

   ```
   DefaultContext ctx = new DefaultContext(con);
   DefaultContext.setDefaultContext(ctx);
   ```
3. Set the execution context (using an ExecutionContext object):

   ```
   ExecutionContext execCtx = ctx.getExecutionContext();
   ```
4. Invoke a stored procedure that returns result sets. In the following example, a stored procedure named GET_HIGH_SALARIES is invoked, and is passed an input variable (called inSalaryThreshold):

   ```
   #sql {CALL GET_HIGH_SALARIES(:in inSalaryThreshold, :out outErrorCode)};
   ```
5. Declare a ResultSet object, and use the ExecutionContext object's getNextResultSet() method to accept result sets from the stored procedure. For multiple result sets, put the getNextResultSet() call in a loop structure. Each result set returned by the stored procedure will spawn a loop iteration. Inside the loop, you can fetch the result set rows method, and then close the result set object (with the ResultSet object's close() method). For example:

   ```
   ResultSet rs = null;

   while ((rs = execCtx.getNextResultSet()) != null)
   {
     ResultSetMetaData stmtInfo = rs.getMetaData();
     int numOfColumns = stmtInfo.getColumnCount();
     int r = 0;

     // Result set rows are fetched and printed to screen.
     while (rs.next())
     {
       r++;
       System.out.print("Row: " + r + ": ");
       for (int i=1; i <= numOfColumns; i++)
       {
         System.out.print(rs.getString(i));
         if (i != numOfColumns)
         {
           System.out.print(", ");
   ```

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

    rs.close();
  }
```

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Returning Result Sets From SQL and Embedded SQL Stored Procedures" on page 38
- "Returning Result Sets From JDBC Stored Procedures" on page 41
- "Returning Result Sets From SQLj Stored Procedures" on page 40

Related samples:

- "SpClient.sqlj -- Call a variety of types of stored procedures from SpServer.sqlj (SQLj)"

Receiving Stored Procedure Result Sets in JDBC Applications and Routines

You can receive result sets from stored procedures you invoke from a JDBC routine or application.

Procedure:

To accept stored procedure result sets from within a JDBC routine or application:

1. Open a database connection (using a Connection object):

   ```
   Connection con =
       DriverManager.getConnection("jdbc:db2:sample", userid, passwd);
   ```

2. Prepare the CALL statement that will invoke a stored procedure that returns result sets (using a CallableStatement object). In the following example, a stored procedure named GET_HIGH_SALARIES is invoked. The prepare is followed by the assignment of an input variable (called inSalaryThreshold -- a numeric value to be passed to the stored procedure) to the value of the parameter marker in the previous statement. (A parameter marker is indicated with a ″?″.)

   ```
   String query = "CALL GET_HIGH_SALARIES(?)";

   CallableStatement stmt = con.prepareCall(query);
   stmt.setDouble(1, inSalaryThreshold);
   ```

3. Call the stored procedure:

```
stmt.execute();
```

4. Use the CallableStatement object's getResultSet() method to accept the first result set from the stored procedure and fetch the rows from the result sets using the fetchAll() method:

```
ResultSet rs = stmt.getResultSet();

// Result set rows are fetched and printed to screen.
while (rs.next())
{
  r++;
  System.out.print("Row: " + r + ": ");
  for (int i=1; i <= numOfColumns; i++)
  {
    System.out.print(rs.getString(i));
    if (i != numOfColumns)
    {
      System.out.print(", ");
    }
  }
  System.out.println();
}
```

5. For multiple result sets, use the CallableStatement object's getNextResultSet() method to enable the following result set to be read. Then repeat the process in the previous step, where the ResultSet object accepts the current result set, and fetches the result set rows. For example:

```
while (callStmt.getMoreResults())
{
  rs = callStmt.getResultSet()

  ResultSetMetaData stmtInfo = rs.getMetaData();
  int numOfColumns = stmtInfo.getColumnCount();
  int r = 0;

  // Result set rows are fetched and printed to screen.
  while (rs.next())
  {
    r++;
    System.out.print("Row: " + r + ": ");
    for (int i=1; i <= numOfColumns; i++)
    {
      System.out.print(rs.getString(i));
      if (i != numOfColumns)
      {
        System.out.print(", ");
      }
    }
    System.out.println();
  }
}
```

6. Close the ResultSet object with its close() method:

```
rs.close();
```

Related concepts:

- "Stored Procedure Result Sets" on page 36

Related tasks:

- "Returning Result Sets From SQL and Embedded SQL Stored Procedures" on page 38
- "Returning Result Sets From JDBC Stored Procedures" on page 41
- "Returning Result Sets From SQLj Stored Procedures" on page 40

Related samples:

- "SpClient.java -- Call a variety of types of stored procedures from SpServer.java (JDBC)"

Parameter Handling in PROGRAM TYPE MAIN or PROGRAM TYPE SUB Stored Procedures

Stored procedures can accept parameters in the style of main routines or subroutines. This is determined when you register your stored procedure with the CREATE PROCEDURE statement.

C or C++ stored procedures of PROGRAM TYPE SUB accept arguments in the same manner as C or C++ subroutines. Pass parameters as pointers. For example, the following C stored procedure signature accepts parameters of type INTEGER, SMALLINT, and CHAR(3):

```
int storproc (sqlint32 *arg1, sqlint16 *arg2, char *arg3)
```

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. The following parameter-style 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[])
```

To write a C 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:

- The stored procedure accepts parameters through two arguments:
  - a parameter counter variable; for example, *argc*
  - an array of pointers to the parameters; for example, *char **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 as defined by the PARAMETER STYLE of 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]*:

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

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

  /* argv[0] contains the stored procedure name. */
  /* 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 */
```

Related concepts:

- “Routines: Stored Procedures” on page 7

Related reference:

- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*

Related samples:

- “spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc ”
- “spserver.sqc -- A variety of types of stored procedures (C)”

UDF and Method Features

Unlike stored procedures, UDFs and methods are invoked from within SQL statements. Where a stored procedure is invoked once when it is CALLed, a function or a method can be invoked multiple times from a single reference in an SQL statement. This difference in implementation requires special features. The following sections describe scratchpads, which can be used to preserve state information between invocations, and the processing model for UDFs and methods registered with the FINAL CALL option.

Scratchpads for UDFs and Methods

A *scratchpad* enables a user-defined function or method to save its state from one invocation to the next. For example, here are two situations where saving state between invocations is beneficial:

1. Functions or methods that, to be correct, depend on saving state.

   An example of such a function or method is a simple counter function that returns a '1' the first time it is called, and increments the result by one each successive call. Such a function could, in some circumstances, be used to number the rows of a SELECT result:

   ```
   SELECT counter(), a, b+c, ...
     FROM tablex
     WHERE ...
   ```

   The function needs a place to store the current value for the counter between invocations, where the value will be guaranteed to be the same for the following invocation. On each invocation, the value can then be incremented and returned as the result of the function.

   This type of routine is NOT DETERMINISTIC. Its output does not depend solely on the values of its SQL arguments.
2. Functions or methods where the performance can be improved by the ability to perform some initialization actions.

   An example of such a function or method, 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 that contain the string 'myocardial infarction' from the document application, that 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 the first call to see if the document ID that is passed as the second argument is contained in the list.

This type of routine is DETERMINISTIC. 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 statement:

```
CREATE FUNCTION counter()
  RETURNS int ... SCRATCHPAD;

CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD 10000;
```

The SCRATCHPAD keyword tells DB2® to allocate and maintain a scratchpad for a routine. The default size for a scratchpad is 100 bytes, but you can determine the size (in bytes) for a scratchpad. The *match* example is 10000 bytes long. DB2 initializes the scratchpad to binary zeros before the first invocation. If the scratchpad is being defined for a table function, and if the table function is also defined 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 after its initialization. For scalar functions defined with scratchpads, DB2 also does not examine or change the scratchpad's content after its initialization. A pointer to the scratchpad is passed to the routine on each invocation, and DB2 preserves the routine's 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, otherwise it could allocate memory for the list and keep the address of the acquired memory in the scratchpad. Scratchpads can be variable length: the length is defined in the CREATE statement for the routine.

The scratchpad only applies to the individual reference to the routine in the statement. If there are multiple references to a routine in a statement, each reference has its own scratchpad, thus scratchpads cannot be used to communicate between references. The scratchpad only applies to a single DB2 agent (an agent is a DB2 entity that performs processing of all aspects of a statement). There is no ″global scratchpad″ to coordinate the sharing of scratchpad information between the agents. This is especially important for situations where DB2 establishes multiple agents to process a statement (in either a single partition or multiple partition database). In these cases, even though there may only be a single reference to a routine in a statement, there could be multiple agents doing the work, and each would have its own scratchpad. In a multiple partition database, where a statement referencing a UDF is processing data on multiple partitions, and invoking the UDF on each partition, the scratchpad would only apply to a single partition. As a result, there is a scratchpad on each partition where the UDF is executed.

If the correct execution of a function depends on there being a single scratchpad per reference to the function, then register the function as DISALLOW PARALLEL. This will force the function to run on a single partition, thereby guaranteeing that only a single scratchpad will exist per reference to the function.

Because it is recognized that a UDF or method may want to acquire system resources, the UDF or method can be defined with the FINAL CALL keyword. This keyword tells DB2 to call the UDF or method at end-of-statement processing so that the UDF or method can release its system resources. It is vital that a routine free any resources it acquires; even a small leak can become a big leak in an environment where the statement is repetitively invoked, and a big leak can cause a DB2 crash.

Since the scratchpad is of fixed size, the UDF or method may want to allocate memory for itself and thus uses the final call to free the memory. For example, the preceding *match* function 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 10000 FINAL CALL;
```

For UDFs or methods that use a scratchpad and are referenced in a subquery, DB2 may decide to make a final call (if the UDF or method is so specified) and refresh the scratchpad between invocations of the subquery. You can protect yourself against this possibility, if your UDFs or methods are ever used in subqueries, by defining the UDF or method with FINAL CALL and using the call-type argument, or by always checking for the *binary zero* state of the scratchpad.

If you do specify FINAL CALL, please note that your UDF or method receives a call of type FIRST. This could be used to acquire and initialize some persistent resource.

Related concepts:

- "Scratchpads on 32-bit and 64-bit Platforms" on page 52
- "Method and Scalar Function Processing Model" on page 53

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Scratchpads on 32-bit and 64-bit Platforms

To make your UDF or method code portable between 32-bit and 64-bit platforms, you must take care in the way you create and use scratchpads that contain 64-bit values. It is recommended that you 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.

A scratchpad is passed in the form of a LOB, which has the structure:

```
struct lob
{
   sqlint32 length;
   char data[100];
}
```

When defining its own structure for the scratchpad, a routine has two choices:

1. Redefine the entire scratchpad LOB, in which case it needs to include an explicit length field. For example:

   ```
   struct spadlob
   {
     sqlint32 lob_length;
     sqlint32 int_var;
     sqlint64 bigint_var;
   };
   void SQL_API_FN routine( ..., struct spadlob* scratchpad, ... )
   {
     /* Use scratchpad */
   }
   ```

2. Redefine just the data portion of the scratchpad LOB, in which case no length field is needed.

```
struct spaddata
{
  sqlint32 int_var;
  sqlint64 bigint_var;
};
void SQL_API_FN routine( ..., struct lob* lob_spad, ... )
{
  struct spaddata* scratchpad = (struct spaddata*)lob_spad-->data;
  /* Use scratchpad */
}
```

Since the application cannot change the value in the length field of the scratchpad LOB, there is no significant benefit to coding the routine as shown in the first example. The second example is also portable between computers with different word sizes, so it is the preferred way of writing the routine.

Related concepts:

- "Scratchpads for UDFs and Methods" on page 49

Method and Scalar Function Processing Model

The processing model for methods and scalar UDFs that are defined with the FINAL CALL specification is as follows:

FIRST call
: This is a special case of the NORMAL call, identified as FIRST to enable the function to perform any initial processing. Arguments are evaluated and passed to the function. Normally, the function will return a value on this call, but it can return an error, in which case no NORMAL or FINAL call is made. If an error is returned on a FIRST call, the method or UDF must clean up before returning, because no FINAL call will be made.

NORMAL call
: These are the second through second-last calls to the function, as dictated by the data and the logic of the statement. The function is expected to return a value with each NORMAL call after arguments are evaluated and passed. If NORMAL call returns an error, no further NORMAL calls are made, but the FINAL call is made.

FINAL call
: This is a special call, made at end-of-statement processing (or CLOSE of a cursor), provided that the FIRST call succeeded. No argument values are passed on a FINAL call. This call is made so that the function can clean up any resources. The function does not return a value on this call, but may return an error.

For methods or scalar UDFs not defined with FINAL CALL, only NORMAL calls are made to the function, which normally returns a value for each call. If a NORMAL call returns an error, or if the statement encounters another error, no more calls are made to the function.

Note: This model describes the ordinary error processing for methods and 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.

Related concepts:

- "Routines: Scalar User-Defined Functions" on page 8
- "Routines: Methods" on page 10

User-Defined Table Functions

In addition to returning scalar values, UDFs can also be developed to return tables. The following sections describe user-defined table functions and the processing model for table UDFs registered with the FINAL CALL option.

User-Defined Table Functions

A user-defined table function delivers a table to the SQL in which it is referenced. A table UDF reference is only valid in a FROM clause of a SELECT statement. 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 calls 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.
- 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 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.

Related reference:

- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*
- “Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL” on page 74

Table Function Processing Model

The processing model for table UDFs that are defined with the FINAL CALL specification is as follows:

FIRST call
: This call is made before the first OPEN call, and its purpose is to enable the function to perform any initial processing. The scratchpad is cleared prior to this call. Arguments are evaluated and passed to the function. The function does not return a row. If the function returns an error, no further calls are made to the function.

OPEN call
: This call is made to enable the function to perform special OPEN processing specific to the scan. The scratchpad (if present) is not cleared prior to the call. Arguments are evaluated and passed. The function does not return a row on an OPEN call. If the function returns an error from the OPEN call, no FETCH or CLOSE call is made, but the FINAL call will still be made at end of statement.

FETCH call
: FETCH calls continue to be made until the function returns the SQLSTATE value signifying end-of-table. It is on these calls that the UDF develops and returns a row of data. Argument values may be passed to the function, but they are pointing to the same values that were passed on OPEN. Therefore, the argument values may not be current and should not be relied upon. If you do need to maintain current values between the invocations of a table function, use a scratchpad. The function can return an error on a FETCH call, and the CLOSE call will still be made.

CLOSE call
: This call is made at the conclusion of the scan or statement, provided

that the OPEN call succeeded. Any argument values will not be current. The function can return an error.

FINAL call
The FINAL call is made at the end of the statement, provided that the FIRST call succeeded. This call is made so that the function can clean up any resources. The function does not return a value on this call, but may return an error.

For table UDFs not defined with FINAL CALL, only OPEN, FETCH, and CLOSE calls are made to the function, which normally returns a value for each call. Before each OPEN call, the scratchpad (if present) is cleared.

The difference between table UDFs that are defined with FINAL CALL and those defined with NO FINAL CALL can be seen when examining a scenario involving a join or a subquery, where the table function access is the ″inner″ access. For example, in a statement such as:

```
SELECT x,y,z,... FROM table_1 as A,
  TABLE(table_func_1(A.col1,...)) as B
  WHERE...
```

In this case, the optimizer would open a scan of table_func_1 for each row of table_1. This is because the value of table_1's col1, which is passed to table_func_1, is used to define the table function scan.

For NO FINAL CALL table UDFs, the OPEN, FETCH, FETCH, ..., CLOSE sequence of calls repeats for each row of table_1. Note that each OPEN call will get a clean scratchpad. Because the table function does not know at the end of each scan whether there will be more scans, it must clean up completely during CLOSE processing. This could be inefficient if there is significant one-time open processing that must be repeated.

FINAL CALL table UDFs, provide a one-time FIRST call, and a one-time FINAL call. These calls are used to amortize the expense of the initialization and termination costs across all the scans of the table function. As before, the OPEN, FETCH, FETCH, ..., CLOSE calls are made for each row of the outer table, but because the table function knows it will get a FINAL call, it does not need to clean everything up on its CLOSE call (and reallocate on subsequent OPEN). Also note that the scratchpad is not cleared between scans, largely because the table function resources will span scans.

At the expense of managing two additional call types, the table UDF may be able to achieve greater efficiency in these join and subquery scenarios. Deciding whether to define the table function as FINAL CALL depends on how it is expected to be used.

Related concepts:

- "Table Function Execution Model for Java" on page 57
- "Routines: Table User-Defined Functions" on page 9

Related reference:

- "CREATE FUNCTION (OLE DB External Table) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (SQL Scalar, Table or Row) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Table) statement" in the *SQL Reference, Volume 2*

Table Function Execution Model for Java

For table functions written in Java™ and using PARAMETER STYLE DB2GENERAL, it is important to understand what happens at each point in DB2's processing of a given statement. The following table details this information for a typical table function. 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.
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.

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
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 that 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 with 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 useful things, but they are supported.

Related concepts:

- "DB2GENERAL Routines" on page 303
- "Java Routines" on page 118
- "Table Function Processing Model" on page 55

Related reference:

- "CREATE FUNCTION (External Table) statement" in the *SQL Reference, Volume 2*

Chapter 3. SQL-Bodied Routines

SQL-bodied routines are composed entirely of SQL statements. You specify these statements in the CREATE statement that you use to register the routine. You can also use the IBM DB2 Development Center to help you register the routine with DB2, specify the source statements for the SQL-bodied routine, and prepare the routine for execution.

CREATE Statements for SQL-Bodied Routines

To issue a CREATE 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 routine 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 -tdterm-char -vf script-name
```

where *term-char* represents the terminating character, and where *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 system command prompt:

```
db2 -td@ -vf script.db2
```

Related concepts:

- "Routines (Stored Procedures, UDFs, Methods)" on page 3

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (SQL Scalar, Table or Row) statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE (SQL) statement" in the *SQL Reference, Volume 2*

Dynamic SQL in SQL-Bodied Routines

SQL routines, like external routines, 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 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 routine, you must include the following statements in the SQL routine body:

1. Declare a variable of type VARCHAR that is large enough to hold your dynamic SQL statement using a DECLARE statement.

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.
3. Create a prepared statement from the statement string using a PREPARE statement.
4. Execute the prepared statement using an EXECUTE statement. If the statement string includes input parameter markers, use the USING clause to replace it with the value of a variable. If the statement includes output parameter markers, use the INTO clause to specify the variables that will receive the output.

Note: Statement names defined in PREPARE statements for SQL routines are treated as scoped variables. Once the SQL routine 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.

The following example shows an SQL procedure that includes dynamic SQL statements:

The SQL procedure receives a department number (*deptNumber*) as an input parameter. In the SQL 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 issues 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
```

Related concepts:

- "Dynamic SQL Support Statements" in the *Application Development Guide: Programming Client Applications*

Related reference:

- "EXECUTE statement" in the *SQL Reference, Volume 2*
- "PREPARE statement" in the *SQL Reference, Volume 2*

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.

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 *instance*/function/routine/sqlproc/*db_name*/*schema_name*/tmp

where *instance* represents the path of the DB2 instance, *db_name* represents the database alias, and *schema_name* represents the schema with which the CREATE PROCEDURE statement was issued.

Windows
instance\function\routine\sqlproc\*db_name**schema_name*\tmp

where *instance* represents the path of the DB2 instance, *db_name* represents the database alias, and *schema_name* represents the schema with which the CREATE PROCEDURE statement was issued.

Note: If the SQL 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
```

Related tasks:

- "Retaining Intermediate Files for SQL Procedures" in the *Application Development Guide: Building and Running Applications*
- "Debugging Routines" on page 31

Related reference:

- "CURRENT SCHEMA special register" in the *SQL Reference, Volume 1*

Condition Handlers in SQL Procedures

The sections that follow describe condition handlers, and how they can be used to enable SQL procedures to react to various database conditions.

Condition Handlers 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 conditions, named conditions, or specific SQLSTATE values.

If a statement in your SQL procedure raises 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 such a condition, DB2 passes control to the next statement in the SQL procedure body. If the SQLCODE and SQLSTATE variables have been declared, they will contain the corresponding values for the condition.

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 the SQLSTATE and SQLCODE variables have been declared, their values after the successful execution of a handler will be '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 caller.

Related concepts:

- "SIGNAL and RESIGNAL Statements in Condition Handlers" on page 67
- "Condition Handler Declarations" on page 64
- "SQLCODE and SQLSTATE Variables in SQL Procedures" on page 68

Related tasks:

- "Displaying Error Messages for SQL Procedures" on page 62

Condition Handler Declarations

In order to define the behavior of your SQL procedure when certain conditions occur, you need to declare 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-types

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.

Conditions
DB2 provides three general conditions:

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 or an SQLSTATE beginning with the characters '02'.

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. The corresponding SQLSTATE value will begin with the characters '01'.

You can also use the DECLARE statement to define your own condition for a specific SQLSTATE.

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.

The following examples demonstrate simple condition handlers:

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 CONTINUE HANDLER FOR NOT FOUND SET at_end = 1;
```

EXIT handler
In this example, the scope of the exit handler is confined to the compound statement labeled A. If the table JAVELIN does not exist, the DROP statement raises the NO_TABLE condition. The exit handler will be activated, OUT_BUFFER will be set to the string, `'Table does not exist'`, and execution will continue with the INSERT statement at C, without visiting any more statements in compound statement A. If the DROP statement completes successfully, the handler will not be activated and execution will continue with the SET statement at B.

```
CREATE PROCEDURE EXIT_TEST ()
 LANGUAGE SQL
 BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
    DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

    A: BEGIN
          DECLARE EXIT HANDLER FOR NO_TABLE
          BEGIN
             SET OUT_BUFFER='Table does not exist';
          END;

          -- Drop potentially nonexistent table:
          DROP TABLE JAVELIN;

       B: SET OUT_BUFFER='Table dropped successfully';
    END;

    -- Copy OUT_BUFFER to some message table:
    C: INSERT INTO MESSAGES VALUES OUT_BUFFER;
 END
```

UNDO handler

In this example, the scope of the undo handler is confined to the compound statement labeled A. If table JAVELIN does not exist, the DROP statement raises the NO_TABLE condition. The undo handler will be activated, the INSERT preceding the DROP will be rolled back, OUT_BUFFER will be set to the string ’`Table does not exist`’, and execution will continue with the INSERT statement at C, without visiting any more statements in compound statement A. If the DROP statement completes successfully, the handler will not be activated and execution will continue with the SET statement at B.

```
CREATE PROCEDURE UNDO_TEST ()
 LANGUAGE SQL
 BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
    DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

    A: BEGIN ATOMIC
          DECLARE UNDO HANDLER FOR NO_TABLE
          BEGIN
             SET OUT_BUFFER='Table does not exist';
          END;

          INSERT INTO MESSAGES VALUES
             'This message will be removed by a rollback.';

          -- Drop potentially nonexistent table:
          DROP TABLE JAVELIN;

       B: SET OUT_BUFFER='Table dropped successfully';
    END;
```

```
      -- Copy OUT_BUFFER to some message table:
      C: INSERT INTO MESSAGES VALUES OUT_BUFFER;
   END
```

Note: You can only declare UNDO handlers in ATOMIC compound statements.

Related concepts:

- "Condition Handlers in SQL Procedures" on page 63
- "SIGNAL and RESIGNAL Statements in Condition Handlers" on page 67
- "SQLCODE and SQLSTATE Variables in SQL Procedures" on page 68

Related reference:

- "Compound SQL (Embedded) statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "Compound SQL (Dynamic) statement" in the *SQL Reference, Volume 2*

SIGNAL and RESIGNAL Statements in Condition Handlers

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 along with the raised SQLSTATE.

In the following example, the SQL procedure body declares a condition handler for the custom SQLSTATE 72822. When the SQL 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 HANDLER FOR SQLSTATE '72822'
BEGIN
  IF ( var = 'OK' )
    RESIGNAL SQLSTATE '72623' SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
  ELSE
    RESIGNAL SQLSTATE '72319' SET MESSAGE_TEXT = var;
END;

SIGNAL SQLSTATE '72822';
```

Related concepts:

- "Condition Handlers in SQL Procedures" on page 63

- “Condition Handler Declarations” on page 64
- “SQLCODE and SQLSTATE Variables in SQL Procedures” on page 68

Related reference:

- “SIGNAL statement” in the *SQL Reference, Volume 2*
- “RESIGNAL statement” in the *SQL Reference, Volume 2*

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';
```

DB2® implicitly sets these variables whenever a statement is executed. If a statement raises a condition for which a handler exists, the values of the SQLSTATE and SQLCODE variables are available at the beginning of the handler execution. However, the variables are reset as soon as the first statement in the handler is executed. Therefore, it is common practice to copy the values of SQLSTATE and SQLCODE into local variables in the first statement of the handler. In the following example, a CONTINUE handler for any condition is used to copy the SQLCODE variable into another variable named `retcode`. The variable `retcode` can then be used in the executable statements to control procedural logic, or pass the value back as an output parameter.

```
BEGIN
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE retcode INTEGER DEFAULT 0;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
     SET retcode = SQLCODE;

  executable-statements
END
```

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.

Related concepts:

- “Condition Handlers in SQL Procedures” on page 63
- “SIGNAL and RESIGNAL Statements in Condition Handlers” on page 67

- “Condition Handler Declarations” on page 64

Chapter 4. External Routines

External routines can be written in the following programming languages: C, C++, Java, and OLE. In addition to these languages, stored procedures can also be written in COBOL.

In order to build an external routine, you need to install and configure the supported compilers/developer kits on the database server, depending on the routine’s language. External routines must be built and registered before you can invoke them.

Parameter Styles for External Routines

Each routine must conform to a particular convention for the exchange of parameters. These conventions are known as *parameter styles*. You assign a particular parameter style to a routine during its registration with the PARAMETER STYLE clause. Following are the available parameter styles and their attributes.

Table 1. Parameter styles

Parameter style	Supported language	Supported routine type	Description
SQL [1]	• C/C++ • OLE • COBOL [2]	• UDFs • stored procedures • methods	In addition to the parameters passed during invocation, the following arguments are passed to the routine in the following order: • A null indicator for each parameter or result declared in the CREATE statement. • The SQLSTATE to be returned to DB2. • The qualified name of the routine. • The specific name of the routine. • The SQL diagnostic string to be returned to DB2. Depending on options specified in the CREATE statement and the routine type, the following arguments can be passed to the routine in the following order: • A buffer for the scratchpad. • The call type of the routine. • The dbinfo structure (contains information about the database).
DB2SQL [1]	• C/C++ • OLE • COBOL	• stored procedures	In addition to the parameters passed during invocation, the following arguments are passed to the stored procedure in the following order: • A vector containing a null indicator for each parameter on the CALL statement. • The SQLSTATE to be returned to DB2. • The qualified name of the stored procedure. • The specific name of the stored procedure. • The SQL diagnostic string to be returned to DB2. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure.
JAVA	• Java™	• UDFs • stored procedures	PARAMETER STYLE JAVA routines use a parameter passing convention that conforms to the Java language and SQLj Routines specification. For stored procedures, INOUT and OUT parameters will be passed as single entry arrays to facilitate the returning of values. In addition to the IN, OUT, and INOUT parameters, Java method signatures for stored procedures include a parameter of type ResultSet[] for each result set specified in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement. For PARAMETER STYLE JAVA UDFs and methods, no additional arguments to those specified in the routine invocation are passed.

Table 1. Parameter styles (continued)

Parameter style	Supported language	Supported routine type	Description
DB2GENERAL	• Java	• UDFs • stored procedures • methods	This type of routine will use a parameter passing convention that is defined for use with Java methods. Unless you are developing table UDFs, UDFs with scratchpads, or need access to the dbinfo structure, it is recommended that you use PARAMETER STYLE JAVA. For PARAMETER STYLE DB2GENERAL routines, no additional arguments to those specified in the routine invocation are passed.
GENERAL	• C/C++ • COBOL	• stored procedures	A PARAMETER STYLE GENERAL stored procedure receives parameters from the CALL statement in the invoking application or routine. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. GENERAL is the equivalent of SIMPLE stored procedures for DB2 Universal Database for OS/390 and z/OS.
GENERAL WITH NULLS	• C/C++ • COBOL	• stored procedures	A PARAMETER STYLE GENERAL WITH NULLS stored procedure receives parameters from the CALL statement in the invoking application or routine. Also included is a vector containing a null indicator for each parameter on the CALL statement. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 Universal Database for OS/390 and z/OS.

Note:

1. For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.
2. COBOL can only be used to develop stored procedures.

Related concepts:

- “DB2GENERAL Routines” on page 303
- “Java Routines” on page 118

Related reference:

- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*
- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*
- “CREATE TYPE (Structured) statement” in the *SQL Reference, Volume 2*
- “CREATE METHOD statement” in the *SQL Reference, Volume 2*
- “Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL” on page 74

Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL

In addition to the SQL arguments that are specified in the DML reference for a routine, DB2 passes additional arguments to the external routine body. The nature and order of these arguments is determined by the parameter style with which you registered your routine. To ensure that information is exchanged correctly between invokers and the routine body, you must ensure that your routine accepts arguments in the order they are passed, according to the parameter style being used. The sqludf include file can aid you in handling and using these arguments.

The following parameter styles are applicable only to LANGUAGE C, LANGUAGE OLE, and LANGUAGE COBOL routines.

Syntax for Passing Arguments to PARAMETER STYLE SQL Routines

```
►►─┬─────────────────────────────────────────────┬─sqlstate──routine-name──────────►
   │ ┌─────────────────┐  ┌───────────────────┐  │
   └─▼─SQL-argument────┴──▼─SQL-argument-ind──┴──┘

►─specific-name──diagnostic-message──┬────────────┬──┬───────────┬──┬────────┬──────►◄
                                     └─scratchpad─┘  └─call-type─┘  └─dbinfo─┘
```

Syntax for Passing Arguments to PARAMETER STYLE DB2SQL Stored Procedures

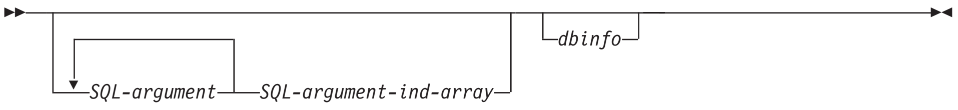
```
►►─┬───────────────────────────────────────────────┬─sqlstate──routine-name────────►
   │ ┌─────────────────┐                           │
   └─▼─SQL-argument────┴───SQL-argument-ind-array──┘

►─specific-name──diagnostic-message──┬────────┬────────────────────────────────────►◄
                                     └─dbinfo─┘
```

Syntax for Passing Arguments to PARAMETER STYLE GENERAL Stored Procedures

```
►►─┬─────────────────────┬──┬────────┬─────────────────────────────────────────────►◄
   │ ┌─────────────────┐ │  └─dbinfo─┘
   └─▼─SQL-argument────┴─┘
```

Syntax for Passing Arguments to PARAMETER STYLE GENERAL WITH NULLS Stored Procedures



Note: For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.

The arguments for the above parameter styles are described as follows:

SQL-argument...

Each *SQL-argument* represents one input or output value defined when the routine was created. The list of arguments is determined as follows:

- For a scalar function, one argument for each input parameter to the function followed by one *SQL-argument* for the result of the function.
- For a table function, one argument for each input parameter to the function followed by one *SQL-argument* for each column in the result table of the function.
- For a method, one *SQL-argument* for the subject type of the method, then one argument for each input parameter to the method followed by one *SQL-argument* for the result of the method.
- For a stored procedure, one *SQL-argument* for each parameter to the stored procedure.

Each *SQL-argument* is used as follows:

- Input parameter of a function or method, subject type of a method, or an IN parameter of a stored procedure

  This argument is set by DB2 before calling the routine. The value of each of these arguments is taken from the expression specified in the routine invocation. It is expressed in the data type of the corresponding parameter definition in the CREATE statement.
- Result of a function or method or an OUT parameter of a stored procedure

  This argument is set by the routine before returning to DB2. DB2 allocates the buffer and passes its address to the routine. The routine puts the result value into the buffer. Enough buffer space is allocated by DB2 to contain the value expressed in the data type. For character types and LOBs, this means the maximum size, as defined in the create statement, is allocated.

  For scalar functions and methods, the result 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, 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, the routine should not return more bytes than is required for the data type and length of the result. Maximums are defined during the creation of the routine's catalog entry. An overwrite by the routine can cause unpredictable results or an abnormal termination.

- INOUT parameter of a stored procedure

  This argument behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The buffer allocated by DB2 for the argument is large enough to contain the maximum size of the data type of the parameter defined in the CREATE PROCEDURE statement. For example, an INOUT parameter of a CHAR type could have a 10 byte varchar going in to the stored procedure, and a 100 byte varchar coming out of the stored procedure. The buffer is set by the stored procedure before returning to DB2.

DB2 aligns the data for *SQL-argument* according to the data type and the server platform.

SQL-argument-ind...

There is an *SQL-argument-ind* for each *SQL-argument* passed to the routine. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is used as follows:

- Input parameter of a function or method, subject type of a method, or an IN parameter of a stored procedure

  This argument is set by DB2 before calling the routine. 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 routine is defined with RETURNS NULL ON NULL INPUT, the routine body does not need to check for a NULL value.

However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the routine should check *SQL-argument-ind* before using the corresponding *SQL-argument*.

- Result of a function or method or an OUT parameter of a stored procedure

  This argument is set by the routine before returning to DB2. This argument is used by the routine to signal if the particular result value is NULL:

  0 or positive
  The result is not NULL.

  negative
  The result is the NULL value.

  Even if the routine is defined with RETURNS NULL ON NULL INPUT, the routine body must set the *SQL-argument-ind* of the result. For example, a divide function could set the result to null when the denominator is zero.

  For scalar functions and methods, DB2 treats a NULL result as an arithmetic error 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

  This is also true if you define the function with the RETURNS NULL ON NULL INPUT option

  For table functions, if the UDF takes advantage of the optimization using the result column list, then only the indicators corresponding to the required columns need be set.

- INOUT parameter of a stored procedure

  This argument behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The *SQL-argument-ind* is set by the stored procedure before returning to DB2.

Each *SQL-argument-ind* takes the form of a SMALLINT value. DB2 aligns the data for *SQL-argument-ind* according to the data type and the server platform.

SQL-argument-ind-array
There is an element in *SQL-argument-ind-array* for each SQL-argument passed to the stored procedure. The *n*th element in *SQL-argument-ind-array* corresponds to the *n*th SQL-argument and indicates whether the *SQL-argument* has a value or is NULL

Each element in *SQL-argument-ind-array* is used as follows:

- IN parameter of a stored procedure

  This element is set by DB2 before calling the routine. 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 stored procedure is defined with RETURNS NULL ON NULL INPUT, the stored procedure body does not need to check for a NULL value. However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the stored procedure should check *SQL-argument-ind* before using the corresponding *SQL-argument*.

- OUT parameter of a stored procedure

  This element is set by the routine before returning to DB2. This argument is used by the routine to signal if the particular result value is NULL:

  0 or positive
  The result is not NULL.

  negative
  The result is the NULL value.

- INOUT parameter of a stored procedure

  This element behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The element of *SQL-argument-ind-array* is set by the stored procedure before returning to DB2.

Each element of *SQL-argument-ind-array* takes the form of a SMALLINT value. DB2 aligns the data for *SQL-argument-ind-array* according to the data type and the server platform.

sqlstate This argument is set by the routine before returning to DB2. It can be used by the routine to signal warning or error conditions. The routine can set this argument to any value. The value ’00000’ means that no warning or error situations were detected. Values that start with ’01’ are warning conditions. Values that start with anything other than ’00’ or ’01’ are error conditions. When the routine is called, the argument contains the value ’00000’.

For error conditions, the routine returns an SQLCODE of -443. For warning conditions, the routine returns an SQLCODE of +462. If the SQLSTATE is 38001 or 38502, then the SQLCODE is -487.

The *sqlstate* takes the form of a CHAR(5) value. DB2 aligns the data for *sqlstate* according to the data type and the server platform.

routine-name

This argument is set by DB2 before calling the routine. It is the qualified function name, passed from DB2 to the routine

The form of the *routine-name* that is passed is:

```
schema.routine
```

The parts are separated by a period. Two examples are:

```
PABLO.BLOOP   WILLIE.FINDSTRING
```

This form enables you to use the same routine body for multiple external routines, and still differentiate between the routines 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 routine name that is passed to the function is OBJ.OP.ROTATE, and it is not obvious if the schema name is OBJ or OBJ.OP.

The *routine-name* takes the form of a VARCHAR(257) value. DB2 aligns the data for *routine-name* according to the data type and the server platform.

specific-name

This argument is set by DB2 before calling the routine. It is the specific name of the routine passed from DB2 to the routine.

Two examples are:

```
WILLIE_FIND_FEB99   SQL9904281052440430
```

This first value is provided by the user in his CREATE statement. The second value is generated by DB2 from the current timestamp when the user does not specify a value.

As with the *routine-name* argument, the reason for passing this value is to give the routine the means of distinguishing exactly which specific routine is invoking it.

The *specific-name* takes the form of a VARCHAR(18) value. DB2 aligns the data for *specific-name* according to the data type and the server platform.

diagnostic-message

This argument is set by the routine before returning to DB2. The routine can use this argument to insert message text in a DB2 message.

When the routine returns either an error or a warning, using the *sqlstate* argument described previously, 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 routine. 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 that will appear depends on the lengths of the other tokens because DB2 may truncate the tokens to conform to the 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 routine should not return more text than will fit in the VARCHAR(70) buffer that is passed to it. An overwrite by the routine can cause unpredictable results or an abend.

DB2 assumes that any message tokens returned from the routine to DB2 are in the same code page as the database. Your routine should ensure that this is the case. If you use the 7-bit invariant ASCII subset, your routine can return the message tokens in any code page.

The *diagnostic-message* takes the form of a VARCHAR(70) value. DB2 aligns the data for *diagnostic-message* according to the data type and the server platform.

scratchpad

This argument is set by DB2 before invoking the UDF or method. It is only present for functions and methods that specified the SCRATCHPAD keyword during registration. 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 0s as follows:
  - For scalar functions and methods, it is initialized before the first call, and not generally looked at or modified by DB2 thereafter.
  - For table functions, the scratchpad is initialized 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 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 routine using the same type as either a CLOB or a BLOB, since the argument passed has the same structure.

Ensure your routine code does not make changes outside of the scratchpad buffer. An overwrite by the routine can cause unpredictable results, an abend, and might not result in a graceful failure by DB2.

If a scalar UDF or method that uses a scratchpad is referenced in a subquery, DB2 might 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 can result in the entire scratchpad structure, including the length field, being improperly aligned.

call-type

This argument, if present, is set by DB2 before invoking the UDF or method. This argument is present for all table functions and for scalar functions and methods that specified FINAL CALL during registration

All the current possible values for *call-type* follow. Your UDF or method should contain a switch or case statement that 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 do not explicitly test for condition C you will have trouble when new possibilities are added.

Notes:

1. For all values of *call-type*, it might be appropriate for the routine to set a *sqlstate* 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 routines. The file contains symbolic defines for the following *call-type* values, which are spelled out as constants.

For scalar functions and methods *call-type* contains:

SQLUDF_FIRST_CALL (-1)
This is the FIRST call to the routine for this statement. The *scratchpad* (if any) is set to binary zeros when the routine is called. All argument values are passed, and the routine should do whatever one-time initialization actions are required. In addition, a FIRST call to a scalar UDF or method is like a NORMAL call, in that it is expected to develop and return an answer.

Note: If SCRATCHPAD is specified but FINAL CALL is not, then the routine 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.

SQLUDF_NORMAL_CALL (0)
This is a NORMAL call. All the SQL input values are passed, and the routine is expected to develop and return the result. The routine may also return *sqlstate* and *diagnostic-message* information.

SQLUDF_FINAL_CALL (1)
This is a FINAL call, that is no *SQL-argument* or *SQL-argument-ind* values are passed, and attempts to examine these values may cause unpredictable results. If a *scratchpad* is also passed, it is untouched from the previous call. The routine is expected to release resources at this point.

SQLUDF_FINAL_CRA (255)
This is a FINAL call, identical to the FINAL call described previously, with one additional characteristic, namely that it is made to routines that are defined as being able to issue SQL, and it is made at such a time that the routine must not issue any SQL except CLOSE cursor. (SQLCODE -396, SQLSTATE 38505) For example, when DB2 is in the middle of COMMIT processing, it can not tolerate new SQL, and any FINAL call issued to a routine at that time would be a 255 FINAL call. Routines that are not defined as containing any level of SQL access

will never receive a 255 FINAL call, whereas routines that do use SQL might be given either type of FINAL call.

Releasing resources

A scalar UDF or method is expected to release resources it has required, for example, memory. If FINAL CALL is specified for the routine, 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:

SQLUDF_TF_FIRST (-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. 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.

SQLUDF_TF_OPEN (-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.

SQLUDF_TF_FETCH (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 *scratchpad* is passed to the UDF, then on entry it is untouched from the previous call.

SQLUDF_TF_CLOSE (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.

SQLUDF_TF_FINAL (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.

SQLUDF_TF_FINAL_CRA (255)

This is a FINAL call, identical to the FINAL call described above, with one additional characteristic, namely that it is made to UDFs which are defined as being able to issue SQL, and it is made at such a time that the UDF must not issue any SQL except CLOSE cursor. (SQLCODE -396, SQLSTATE 38505) For example, when DB2 is in the middle of COMMIT processing, it can not tolerate new SQL, and any FINAL call issued to a UDF at that time would be a 255 FINAL call. Note that UDFs which are not defined as containing any level of SQL access will never receive a 255 FINAL call, whereas UDFs which do use SQL may be given either type of FINAL call.

Releasing resources

Write routines 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: 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.

The *call-type* takes the form of an INTEGER value. DB2 aligns the data for *call-type* according to the data type and the server platform.

dbinfo This argument is set by DB2 before calling the routine. It is only present if the CREATE statement for the routine specifies the DBINFO keyword. The argument is the `sqludf_dbinfo` structure defined in the header file `sqludf.h`. The variables in this structure that contain names and identifiers may be longer than the longest value possible in this release of DB2, but they 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. Database name length (dbnamelen)

   The length of *data base name* below. This field is an unsigned short integer.
2. Database 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 previously 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 previously 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 routine 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 previously, identifies the actual length of this field. The *schema name* field described previously, 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, it is 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 Universal Database for z/OS or OS/390
ARI	SQL/DS or DB2 for VM or VSE
QSQ	DB2 Universal Database for iSeries
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. Reserved field (resd0)

    This field is for future use.

14. Platform (platform)

    The operating platform for the application server, as follows:

SQLUDF_PLATFORM_AIX	AIX
SQLUDF_PLATFORM_HP	HP-UX
SQLUDF_PLATFORM_LINUX	Linux
SQLUDF_PLATFORM_MVS	OS/390
SQLUDF_PLATFORM_NT	Windows NT, Windows 2000, Windows XP
SQLUDF_PLATFORM_SUN	Solaris Operating Environment
SQLUDF_PLATFORM_WINDOWS95	Windows 95, Windows 98, Windows Me
SQLUDF_PLATFORM_UNKNOWN	Unknown platform

    For additional platforms that are not contained in the above list, see the contents of the `sqludf.h` file.

15. 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.

16. Reserved field (resd1)

    This field is for future use.

17. Routine id of the stored procedure that invoked the current routine (procid)

    The stored procedure's routine id matches the ROUTINEID column in SYSCAT.ROUTINES, which can be used to retrieve the name of the invoking stored procedure. This field is a 32-bit signed integer.

18. Reserved field (resd2)

    This field is for future use.

19. Table function column list (tfcolumn)

    If this is a table function, this field is a pointer to an array of short integers that is dynamically allocated by DB2. If this is any other type of routine, 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 that 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.

20. Unique application identifier (appl_id)

    This field is a pointer to a C null-terminated string that uniquely identifies the application’s connection to DB2. It is generated by DB2 at connect time.

    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
    ```

21. Reserved field (resd3)

    This field is for future use.

Related concepts:

- “Parameter Styles for External Routines” on page 71
- “Include File for C/C++ Routines (sqludf.h)” on page 102
- “C/C++ Routines” on page 97

Related tasks:

- “Writing Routines” on page 29

Related reference:

- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*
- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*

- “CREATE TYPE (Structured) statement” in the *SQL Reference, Volume 2*
- “Application ID” in the *System Monitor Guide and Reference*
- “CREATE METHOD statement” in the *SQL Reference, Volume 2*

SQL in External Routines

All routines written in an external programming language (such as C or Java) can contain SQL.

The CREATE statements for each routine type (and in the case of methods, CREATE TYPE) define the level of SQL access for a given routine. Based on the nature of the SQL included in your routine, you must choose the applicable clause:

NO SQL
: the routine contains no SQL at all

CONTAINS SQL
: Contains SQL, but neither reads nor writes data (for example: SET SPECIAL REGISTER).

READS SQL DATA
: Contains SQL that may read from tables (SELECT, VALUES statements), but does not modify table data.

MODIFIES SQL DATA
: Contains SQL that updates tables, either user tables directly (INSERT, UPDATE, DELETE statements) or DB2’s catalog tables implicitly (DDL statements). This clause is only applicable to stored procedures.

DB2® will validate at execution time that a routine does not exceed its defined level. For example, if a routine defined as CONTAINS SQL tries to SELECT from a table, an error (SQLCODE -579, SQLSTATE 38004) will result. Also note that for nested references, only the same or more restrictive SQL levels are allowed. For example, MODIFY routines can invoke READ routines, but READ routines cannot invoke MODIFY routines.

A routine executes SQL statements within the database connection scope of the calling application. A routine cannot establish its own connection, nor can it reset the calling application’s connection (SQLCODE -751, SQLSTATE 38003).

Only a stored procedure defined as MODIFIES SQL DATA can issue COMMIT and ROLLBACK statements. Other types of routines (UDFs and methods) cannot issue COMMITs or ROLLBACKs (SQLCODE -751, SQLSTATE 38003). Even though a stored procedure defined as MODIFIES SQL DATA can attempt to COMMIT or ROLLBACK a transaction, it is recommended that a

COMMIT or ROLLBACK be done from the calling application so changes are not unexpectedly committed. Stored procedures cannot issue COMMIT or ROLLBACK statements if the stored procedure was invoked from an application that established a type 2 connection to the database.

Also, only stored procedures defined as MODIFIES SQL DATA can establish their own savepoints, and rollback their own work within the savepoint. Other types of routines (UDFs and methods) cannot establish their own savepoints. A savepoint created within a stored procedure is not released when the stored procedure completes. The application will be able to roll back the savepoint. Similarly, a stored procedure could roll back a savepoint defined in the application. DB2 will implicitly release any savepoints established by the routine when it exits.

A routine may inform DB2 about whether it has succeeded by assigning an SQLSTATE value to the sqlstate argument that DB2 passes to it. Some parameter styles (PARAMETER STYLEs JAVA, GENERAL, and GENERAL WITH NULLS) do not support the exchange of SQLSTATE values.

If, in handling the SQL issued by a routine, DB2 encounters an error, it returns that error to the routine, just as it does for any application. For normal user errors, the routine has an opportunity to take alternative or corrective action. For example, if a routine is trying to INSERT to a table and gets a duplicate key error (SQLCODE -813), it may instead decide to UPDATE the existing row of the table.

There are, however, certain more serious errors that can occur that make it impossible for DB2 to proceed in a normal fashion. Examples of these include deadlock, or database partition failure, or user interrupt. Some of these errors are propagated up to the calling application. Other severe errors that are unit of work related go all the way out to either (a) the application, or (b) a stored procedure that is permitted to issue transaction control statements (COMMIT or ROLLBACK), whichever occurs first in backing out.

If one of these errors occurs during the execution of SQL issued by a routine, the error is returned to the routine, but DB2 remembers that a serious error has occurred. Additionally, in this case, DB2 will automatically fail (SQLCODE -20139, SQLSTATE 51038) any subsequent SQL issued by this routine and by any calling routines. The only exception to this is if the error only backs out to the outermost stored procedure that is permitted to issue transaction control statements. In this case, this stored procedure can continue to issue SQL.

Routines can issue both static and dynamic SQL, and in either case they must be precompiled and bound if embedded SQL is used. For static SQL, the information used in the precompile/bind process is the same as it is for any client application using embedded SQL. For dynamic SQL, you can use the

DYNAMICRULES precompile/bind option to control the current schema and current authentication ID for embedded dynamic SQL. This behavior is different for routines and applications.

The isolation level defined for the routine packages or statements is respected. This can result in a routine running at a more restrictive, or a more generous, isolation level than the calling application. This is important to consider when calling a routine that has a less restrictive isolation level than the calling statement. For example, if a cursor stability function is called from a repeatable read application, the UDF may exhibit non-repeatable read characteristics.

The invoking application or routine is not affected by any changes made by the routine to special register values. Updatable special registers are inherited by the routine from the invoker. Changes to updatable special registers are not passed back to the invoker. Non-updatable special registers get their default value. For further details on updatable and non-updatable special registers, see the related topic, "Special registers".

Routines can OPEN, FETCH, and CLOSE cursors in the same manner as client applications. Multiple invocations (for example, in the case of recursion) of the same function each get their own instance of the cursor. UDFs and methods must close their cursors before the invoking statement completes, otherwise an error will occur (SQLCODE -472, SQLSTATE 24517). The final call for a UDF or method is a good time to close any cursors that remain open. Any opened cursors not closed before completion in a stored procedure are returned to the client application or calling routine as result sets.

Arguments passed to routines are not automatically treated as host variables. This means for a routine to use a parameter as a host variable in its SQL, it must declare its own host variable and copy the parameter value to this host variable.

Note: Embedded SQL routines must be precompiled and bound with the DATETIME option set to ISO.

Related tasks:

- "Customizing Precompile and Bind Options for SQL Procedures" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*

- “CREATE FUNCTION (OLE DB External Table) statement” in the *SQL Reference, Volume 2*
- “CREATE FUNCTION (SQL Scalar, Table or Row) statement” in the *SQL Reference, Volume 2*
- “CREATE FUNCTION (External Scalar) statement” in the *SQL Reference, Volume 2*
- “CREATE FUNCTION (External Table) statement” in the *SQL Reference, Volume 2*
- “CREATE FUNCTION (Sourced or Template) statement” in the *SQL Reference, Volume 2*
- “SQL statements allowed in routines” in the *SQL Reference, Volume 1*
- “CREATE PROCEDURE (External) statement” in the *SQL Reference, Volume 2*
- “CREATE PROCEDURE (SQL) statement” in the *SQL Reference, Volume 2*
- “Special registers” in the *SQL Reference, Volume 1*

Authorizations and Binding for External Routines that Contain SQL

When discussing authorization it is useful to distinguish some roles for the routine and to mention some of the factors that influence the determination of which ID assumes the role:

- Definer: The ID that performs the CREATE statement to register the routine.
- Package owner: The owner of a package that participates in the implementation of a routine. It can be the ID that actually performs the BIND, or if the OWNER precompile/bind option is used, another ID.
- Invoker: The ID that invokes the routine. For dynamic SQL, it can be the runtime authorization ID of the immediately higher-level routine or application (this ID depends on the DYNAMICRULES option with which the higher-level routine/application was bound). For static SQL, it can be the value of the OWNER precompile/bind option of the package that contains the reference to the routine.

  For example, if this referencing package is an application running dynamic SQL and was bound with DYNAMICRULES BIND, then its run-time authorization ID will be its package owner, not the person invoking the package. Also, the package owner will be the actual binder or the value of the OWNER precompile/bind option. In this case, the invoker of the routine assumes this value rather than the ID of the user who is executing the application.

Notes:

1. For static SQL within a routine, the package owner's privileges must be adequate for the SQL actions. These actions include table access, and the execute privilege on any (nested) references to routines.
2. For dynamic SQL within a routine, the userid whose privileges will be validated are governed by the DYNAMICRULES option of the BIND of the routine body.
3. The routine package owner must GRANT EXECUTE on the package to the routine definer. This can be done before or after the routine is registered, but it must be done before the routine is invoked.
4. The routine definer is automatically given EXECUTE WITH GRANT privilege on the routine; the definer must GRANT EXECUTE on the routine to PUBLIC or to any users who are intended to use the routine.
5. It is the definer's authorization to use the packages of a routine that is checked, not the invoker's. Thus the definer, in a sense, encapsulates the privilege of running any packages associated with the routine. It is at package load time that the definer's EXECUTE privilege on the package is verified. This is true for each package associated with the routine.

To correctly create and use an external routine that contains SQL:

1. Definer performs the appropriate CREATE statement to register the routine. This defines the routine to DB2® with its intended level of SQL access, establishes the routine signature, and also points to the routine executable, so the definer needs to be effectively communicating with the package owners and authors of the routine programs. By virtue of a successful CREATE statement, definer has EXECUTE WITH GRANT privilege on the routine.
2. Definer must grant EXECUTE privilege on the routine to any users who are to be permitted use of the routine. (If the package for this routine will recursively call this routine, then this step must be done before the next step.)
3. Package owners precompile and bind the routine programs, or have it done on their behalf. By virtue of successful precompile and bind, the package owners are each given EXECUTE privilege on the respective packages. This step follows step one in this list only to cover the possibility of SQL recursion in the routine. If such recursion does not exist in any particular case, the precompile/bind could precede the CREATE statement.
4. The package owners each must grant EXECUTE privilege on their respective packages to the definer of the routine. This step must come at some time after the previous step.
5. Static usage of the routine: the bind owner of the package referencing the routine must have been given EXECUTE privilege on the routine, so the

previous step must be completed at this point. When the routine executes, DB2 verifies that the definer has the EXECUTE privilege on any package that is needed, so step 3 must be completed for each such package.

6. Dynamic usage of the routine: the authorization ID as controlled by the DYNAMICRULES option for the invoking application must have EXECUTE privilege on the routine (step 4), and the definer of the routine must have the EXECUTE privilege on the packages (step 3).

Related concepts:
- "Privileges, authorities, and authorization" in the *Administration Guide: Implementation*
- "Effects of DYNAMICRULES on Dynamic SQL" on page 94
- "Procedure, function, and method privileges" in the *Administration Guide: Implementation*

Related reference:
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*

Effects of DYNAMICRULES on Dynamic SQL

The PRECOMPILE and BIND option DYNAMICRULES determines what values apply at run-time for the following dynamic SQL attributes:
- The authorization ID that is used during authorization checking.
- The qualifier that is used for qualification of unqualified objects.
- Whether the package can be used to dynamically prepare the following statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE statements.

In addition to the DYNAMICRULES value, the run-time environment of a package controls how dynamic SQL statements behave at run-time. The two possible run-time environments are:
- The package runs as part of a stand-alone program
- The package runs within a routine context

The combination of the DYNAMICRULES value and the run-time environment determine the values for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The four behaviors are:

Run behavior DB2® uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization checking of dynamic SQL statements and for the initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

Bind behavior At run-time, DB2 uses all the rules that apply to static SQL for authorization and qualification. That is, take the authorization ID of the package owner as the value to be used for authorization checking of dynamic SQL statements and the package default qualifier for implicit qualification of unqualified object references within dynamic SQL statements.

Define behavior

Define behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES DEFINEBIND or DYNAMICRULES DEFINERUN. DB2 uses the authorization ID of the routine definer (not the routine's package binder) as the value to be used for authorization checking of dynamic SQL statements and for implicit qualification of unqualified object references within dynamic SQL statements within that routine.

Invoke behavior

Invoke behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES INVOKEBIND or DYNAMICRULES INVOKERUN. DB2 uses the current statement authorization ID in effect when the routine is invoked as the value to be used for authorization checking of dynamic SQL and for implicit qualification of unqualified object references within dynamic SQL statements within that routine. This is summarized by the following table:

Invoking Environment	**ID Used**
Any static SQL	Implicit or explicit value of the OWNER of the package the SQL invoking the routine came from.
Used in definition of view or trigger	Definer of the view or trigger.
Dynamic SQL from a run behavior package	ID used to make the initial connection to DB2.
Dynamic SQL from a define behavior package	Definer of the routine that uses the package that the SQL invoking the routine came from.
Dynamic SQL from an invoke behavior package	Current® authorization ID invoking the routine.

The following table shows the combination of the DYNAMICRULES value and the run-time environment that yields each dynamic SQL behavior.

Table 2. How DYNAMICRULES and the Run-Time Environment Determine Dynamic SQL Statement Behavior

DYNAMICRULES Value	**Behavior of Dynamic SQL Statements in a Standalone Program Environment**	**Behavior of Dynamic SQL Statements in a Routine Environment**
BIND	Bind behavior	Bind behavior
RUN	Run behavior	Run behavior
DEFINEBIND	Bind behavior	Define behavior
DEFINERUN	Run behavior	Define behavior
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

The following table shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

Table 3. Definitions of Dynamic SQL Statement Behaviors

Dynamic SQL Attribute	**Setting for Dynamic SQL Attributes: Bind Behavior**	**Setting for Dynamic SQL Attributes: Run Behavior**	**Setting for Dynamic SQL Attributes: Define Behavior**	**Setting for Dynamic SQL Attributes: Invoke Behavior**
Authorization ID	The implicit or explicit value of the OWNER BIND option	ID of User Executing Package	Routine definer (not the routine's package owner)	Current statement authorization ID when routine is invoked.
Default qualifier for unqualified objects	The implicit or explicit value of the QUALIFIER BIND option	CURRENT SCHEMA Special Register	Routine definer (not the routine's package owner)	Current statement authorization ID when routine is invoked.
Can execute GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE	No	Yes	No	No

Table 3. Definitions of Dynamic SQL Statement Behaviors (continued)

Dynamic SQL Attribute	**Setting for Dynamic SQL Attributes: Bind Behavior**	**Setting for Dynamic SQL Attributes: Run Behavior**	**Setting for Dynamic SQL Attributes: Define Behavior**	**Setting for Dynamic SQL Attributes: Invoke Behavior**

Related concepts:

- "Authorization Considerations for Dynamic SQL" in the *Application Development Guide: Programming Client Applications*

C/C++ Routines

The following sections describe how to write C or C++ routines.

C/C++ Routines

When developing routines in C or C++, it is strongly recommended that you register them using the PARAMETER STYLE SQL clause in the CREATE statement. It is also recommended that you use the sqludf.h include file. It contains structures, definitions and values useful when writing both UDFs and stored procedures.

C/C++ UDFs and Methods:

The C/C++ signature of PARAMETER STYLE SQL UDFs and methods follows this format:

```
SQL_API_RC SQL_API_FN function-name ( SQL-arguments,
                                      SQL-argument-inds,
                                      SQLUDF_TRAIL_ARGS )
```

`SQL_API_RC SQL_API_FN`
: `SQL_API_RC` and `SQL_API_FN` are macros that specify the return type and calling convention for a C/C++ function, which can vary across supported operating systems. They are declared in sqlsystm.h. This macro is required when you write C/C++ routines.

function-name
: Name of the C/C++ function. During routine registration, this value is specified with the library name in the EXTERNAL NAME clause of the CREATE PROCEDURE statement. For C++ routines, the C++ compiler applies type decoration to the entry point name. Either the type decorated name needs to be specified in the EXTERNAL NAME clause, or the entry point should be defined as `extern "C"` in the user code.

SQL-arguments
: Corresponds to the list of input parameters in the routine's CREATE statement.

SQL-argument-inds
: For every SQL-argument there is an indicator variable. Define each indicator with the SQLUDF_NULLIND type definition from sqludf.h.

`SQLUDF_TRAIL_ARGS`
: A macro defined in sqludf.h that defines the trailing arguments for a routine. This includes pointers to the SQLSTATE, fully qualified function name, function specific name, and message text. If your UDF is registered with SCRATCHPAD and FINAL CALL, use the `SQLUDF_TAIL_ARGS_ALL` macro. In addition to the arguments included in `SQLUDF_TRAIL_ARGS`, it contains pointers to the scratchpad, and call type.

The following is an example of a C/C++ UDF that returns the product of its two input arguments:

```
SQL_API_RC SQL_API_FN product ( SQLUDF_DOUBLE *in1,
                                SQLUDF_DOUBLE *in2,
                                SQLUDF_DOUBLE *outProduct,
                                SQLUDF_NULLIND *in1NullInd,
                                SQLUDF_NULLIND *in2NullInd,
                                SQLUDF_NULLIND *productNullInd,
                                SQLUDF_TRAIL_ARGS )
{
  *outProduct = (*in1) * (*in2);

  return (0);
}
```

The corresponding CREATE FUNCTION statement for this UDF is as follows:

```
CREATE FUNCTION product( double in1, double in2 )
  RETURNS double
  LANGUAGE c
  PARAMETER STYLE sql
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'c_rtns!product'
```

The preceding statement assumes that the C/C++ function is in a library called c_rtns.

C/C++ Stored Procedures:

The C/C++ signature of PARAMETER STYLE SQL stored procedures follows this format:

```
SQL_API_RC SQL_API_FN function-name ( SQL-arguments,
                                      SQL-argument-inds,
                                      sqlstate,
                                      routine-name,
                                      specific-name,
                                      diagnostic-message )
```

`SQL_API_RC SQL_API_FN`
: `SQL_API_RC` and `SQL_API_FN` are macros that specify the return type and calling convention for a C/C++ function, which can vary across supported operating systems. They are declared in sqlsystm.h. This macro is required when you write C/C++ routines.

function-name
: Name of the C/C++ function. During routine registration, this value is specified with the library name in the EXTERNAL NAME clause of the CREATE PROCEDURE statement. For C++ routines, the C++ compiler applies type decoration to the entry point name. Either the type decorated name needs to be specified in the EXTERNAL NAME clause, or the entry point should be defined as external ″C″ in the user code.

SQL-arguments
: Corresponds to the list of input parameters in the CREATE PROCEDURE statement. OUT or INOUT mode parameters are passed as single-element arrays.

sqlstate Used by the routine to signal warning or error conditions.

routine-name
: The qualified function name. This value is generated by DB2® and passed to the routine in the form `schema.routine`. This value corresponds to the ROUTINESCHEMA and ROUTINENAME columns in the SYSCAT.ROUTINES view.

specific-name
: The specific function name. This value is generated by DB2 and passed to the routine. This value corresponds to the SPECIFICNAME column in the SYSCAT.ROUTINES view.

diagnostic-message
: Used by the routine to return message text to the invoking application or routine.

Note: Unlike the function signature presented in the C/C++ UDF and Methods section, the function signature presented for C/C++ Stored Procedures does not make use of macros declared in sqludf.h. It is, however, possible to write C/C++ stored procedures with the sqludf.h macros. Conversely, it is also possible to write C/C++ UDFs and methods without the sqludf.h macros.

The following is an example of a C/C++ stored procedure that accepts an input parameter, and then returns an output parameter and a result set:

```
SQL_API_RC SQL_API_FN cstp ( sqlint16 *inParm,
                             double *outParm,
                             sqlint16 *inParmNullInd,
                             sqlint16 *outParmNullInd,
                             char sqlst[6],
                             char qualname[28],
                             char specname[19],
                             char diagmsg[71] )
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL BEGIN DECLARE SECTION;
    sqlint16 sql_inParm;
  EXEC SQL END DECLARE SECTION;

  sql_inParm = *inParm;

  EXEC SQL DECLARE cur1 CURSOR FOR
    SELECT value
    FROM table01
    WHERE index = :sql_inParm;

  *outParm = (*inParm) + 1;

  EXEC SQL OPEN cur1;

  return (0);
}
```

The corresponding CREATE PROCEDURE statement for this stored procedure is as follows:

```
CREATE PROCEDURE cproc( IN inParm INT, OUT outParm INT )
  LANGUAGE c
  PARAMETER STYLE sql
  DYNAMIC RESULT SETS 1
  FENCED THREADSAFE
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'c_rtns!cstp'
```

The preceding statement assumes that the C/C++ function is in a library called c_rtns.

Note: When registering a C or C++ routine on Windows® operating systems, take the following precaution when identifing a routine body in the

CREATE statement's EXTERNAL NAME clause. If you use an absolute path id to identify the routine body, you must append the .dll extension. For example:

```
CREATE PROCEDURE getSalary( IN inParm INT, OUT outParm INT )
  LANGUAGE c
  PARAMETER STYLE sql
  DYNAMIC RESULT SETS 1
  FENCED THREADSAFE
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'd:\mylib\myfunc.dll'
```

Related concepts:

- "Database Manager Instances" in the *Application Development Guide: Building and Running Applications*
- "AIX Export Files for Routines" in the *Application Development Guide: Building and Running Applications*
- "AIX Routines and the CREATE Statement" in the *Application Development Guide: Building and Running Applications*
- "Include File for C/C++ Routines (sqludf.h)" on page 102
- "SQL Data Type Handling in C/C++ Routines" on page 106

Related tasks:

- "Building C Routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building C/C++ Routines on Windows" in the *Application Development Guide: Building and Running Applications*
- "Building C Routines on HP-UX" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on HP-UX" in the *Application Development Guide: Building and Running Applications*
- "Building C Routines on Linux" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on Linux" in the *Application Development Guide: Building and Running Applications*
- "Building C Routines on Solaris" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Routines on Solaris" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "C/C++ Samples" in the *Application Development Guide: Building and Running Applications*
- "Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL" on page 74

Related samples:
- "spserver.c -- Definition of various types of stored procedures (CLI)"
- "udfcli.c -- How to work with different types of user-defined functions (UDFs) (CLI)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "udfemsrv.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)"
- "udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"

Include File for C/C++ Routines (sqludf.h)

The `sqludf.h` include file contains structures, definitions, and values that are useful when writing routines. Although this file has 'udf' in its name, (for historical reasons) it is also useful for stored procedures and methods. When compiling your routine, you need to reference the directory that 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 that 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 routine 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 that is defined with the AS LOCATOR clause. This is applicable only to UDFs and methods.

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. Also included are definitions for referencing these constructs, and the various valid SQLSTATE values.
5. Macros for testing whether the SQL arguments are null.

A corresponding include file for COBOL exists: `sqludf.cbl`. This file only includes definitions for the scratchpad and dbinfo structures.

Related concepts:

- "SQL Data Type Handling in C/C++ Routines" on page 106
- "C/C++ Routines" on page 97

Related reference:

- "Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL" on page 74
- "Supported SQL Data Types in C/C++" on page 103

Supported SQL Data Types in C/C++

The following table lists the supported mappings between SQL data types and C data types for routines. Accompanying each C/C++ data type is the corresponding defined type from sqludf.h.

Table 4. SQL Data Types Mapped to C/C++ Declarations

SQL Column Type	C/C++ Data Type	SQL Column Type Description
SMALLINT	sqlint16 SQLUDF_SMALLINT	16-bit signed integer
INTEGER	sqlint32 SQLUDF_INTEGER	32-bit signed integer
BIGINT	sqlint64 SQLUDF_BIGINT	64-bit signed integer
REAL FLOAT(*n*) where 1<=n<=24	float SQLUDF_REAL	Single-precision floating point
DOUBLE FLOAT FLOAT(*n*) where 25<=n<=53	double SQLUDF_DOUBLE	Double-precision floating point
DECIMAL(*p*, *s*)	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.

Table 4. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
CHAR(*n*)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254 SQLUDF_CHAR	Fixed-length, null-terminated character string
CHAR(*n*) FOR BIT DATA	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254 SQLUDF_CHAR	Fixed-length, null-terminated character string
VARCHAR(*n*)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=32 672 SQLUDF_VARCHAR	Null-terminated varying length string

VARCHAR(*n*) FOR BIT DATA	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
	
	SQLUDF_VARCHAR_FBD
		Not null-terminated varying length character string

LONG VARCHAR	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 700
	
	SQLUDF_LONG
		Not null-terminated varying length character string

CLOB(*n*)	struct {
	sqluint32 length;
	char data[*n*];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_CLOB
		Not null-terminated varying length character string with 4-byte string length indicator

BLOB(*n*)	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_BLOB
		Not null-terminated varying binary string with 4-byte string length indicator

Table 4. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
DATE	char[11] SQLUDF_DATE	Null-terminated character string of the following format: `yyyy-mm-dd`
TIME	char[9] SQLUDF_TIME	Null-terminated character string of the following format: `hh.mm.ss`
TIMESTAMP	char[27] SQLUDF_STAMP	Null-terminated character string of the following format: `yyyy-mm-dd-hh.mm.ss.nnnnnn`
LOB LOCATOR	sqluint32 SQLUDF_LOCATOR	32-bit signed integer

DATALINK	struct {
	   sqluint32 version;
	   char linktype[4];
	   sqluint32 url_length;
	   sqluint32 comment_length;
	   char reserve2[8];
	   char url_plus_comment[230];
	}
	
	SQLUDF_DATALINK	

Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.		
GRAPHIC(*n*)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=127 SQLUDF_GRAPH	Fixed-length, null-terminated double-byte character string
VARGRAPHIC(*n*)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336 SQLUDF_GRAPH	Not null-terminated, variable-length double-byte character string

LONG VARGRAPHIC	struct {
	   sqluint16 length;
	   sqldbchar[*n*]
	}
	
	1<=*n*<=16 350
	
	SQLUDF_LONGVARG
		Not null-terminated, variable-length double-byte character string

Table 4. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description

DBCLOB(*n*)	struct {
	   sqluint32 length;
	   sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
	
	SQLUDF_DBCLOB
		Not null-terminated varying length character string with 4-byte string length indicator

Related concepts:

- "Include File for C/C++ Routines (sqludf.h)" on page 102
- "SQL Data Type Handling in C/C++ Routines" on page 106
- "C/C++ Routines" on page 97

SQL Data Type Handling in C/C++ Routines

This section identifies the valid types for routine parameters and results, and it specifies how the corresponding argument should be defined in your C or C++ language routine. All arguments in the routine must be passed as pointers to the appropriate data type. 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 type required for BIGINT representation between different compilers.

It is the data type for each parameter defined in the routine's CREATE statement that governs the format for argument values. Promotions from the argument's data type may be needed to get the value in the appropriate format. Such promotions are performed automatically by DB2® on argument values. However, if incorrect data types are specified in the routine code, then unpredictable behavior, such as loss of data or abends, will occur.

For the result of a scalar function or method, 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 routine 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 routine must be written to generate a SMALLINT, as defined later in this section. Note that the CAST FROM data type must be *castable* to the RETURNS data type, therefore, it is not possible to arbitrarily choose another data type.

The following is a list of the SQL types and their C/C++ language representations. It includes information on whether each type is valid as a parameter or a result. Also included are examples of how the types could appear as an argument definition in your C or C++ language routine:

- SMALLINT

  Valid. Represent in C as `SQLUDF_SMALLINT` or `sqlint16`.

  Example:

  ```
  sqlint16    *arg1;          /* example for SMALLINT */
  ```

  When defining integer routine parameters, consider using INTEGER rather than SMALLINT because DB2 does not promote INTEGER arguments to SMALLINT. For example, suppose you define a UDF as follows:

  ```
  CREATE FUNCTION SIMPLE(SMALLINT)...
  ```

  If you invoke the SIMPLE function using INTEGER data, (`... SIMPLE(1)...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not perceive the reason for the message. In the preceding 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 `SQLUDF_INTEGER` or `sqlint32`. You must `#include sqludf.h` or `#include sqlsystm.h` to pick up this definition.

  Example:

  ```
  sqlint32 *arg2;          /* example for INTEGER */
  ```
- BIGINT

  Valid. Represent in C as `SQLUDF_BIGINT` or `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` or `#include sqlsystm.h` to pick up the definition.
- REAL or FLOAT(*n*) where 1 <= *n* <= 24

  Valid. Represent in C as `SQLUDF_REAL` or `float`.

  Example:

```
     float *result;         /* example for REAL */
```

- DOUBLE or DOUBLE PRECISION or FLOAT or FLOAT(*n*) where 25 <= *n* <= 53

  Valid. Represent in C as `SQLUDF_DOUBLE` or `double`.

  Example:

  ```
       double  *result;       /* example for DOUBLE   */
  ```

- 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.

  Example:

  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 preceding 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 that explicit casting is required because DECIMAL arguments are not promotable to VARCHAR.

  An advantage of using floating point parameters is that it is easy to perform arithmetic on the values in the routine; 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.

- CHAR(n) or CHARACTER(n) with or without the FOR BIT DATA modifier.

  Valid. Represent in C as `SQLUDF_CHAR` or `char...[n+1]` (this is a C null-terminated string).

  Example:

  ```
  char     arg1[14];      /* example for CHAR(13)   */
  char    *arg1;          /* also acceptable */
  ```

  For a CHAR(n) parameter (with or without FOR BIT DATA), DB2 always moves *n* bytes of data to the buffer and sets the *n+1* byte to null (X'00'). For a RETURNS CHAR(n) value not specified as FOR BIT DATA or an output parameter of a stored procedure, DB2 will look for a null CHAR embedded in the first *n* bytes. If one is found, DB2 will pad from that point through to *n* bytes with blanks.

  If FOR BIT DATA is specified, exercise caution about using the normal C string handling functions in the routine. 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 routine parameters, consider using VARCHAR rather than CHAR as DB2 does not promote VARCHAR arguments to CHAR and string literals are automatically considered as VARCHARs. 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 might not perceive the reason for the message. In the preceding 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 VARCHAR(n) FOR BIT DATA in C as `SQLUDF_VARCHAR_FBD`. Represent LONG VARCHAR in C as `SQLUDF_LONG`. Otherwise represent these two SQL types in C as a structure similar to the following from the sqludf.h include file:

  ```
  struct sqludf_vc_fbd
  {
     unsigned short length;         /* length of data */
     char           data[1];        /* first char of data */
  };
  ```

The [1] 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 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 routine for parameters using the structure variable `length`. For the RETURNS clause, the length that is passed to the routine is the length of the buffer. What the routine 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 `SQLUDF_VARCHAR` or `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 string. The C string-handling functions are thus well suited for manipulation of these values. For a RETURNS VARCHAR(n) value or an output parameter of a stored procedure, the routine 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 acceptable */
  ```

- DATE

  Valid. Represent in C same as `SQLUDF_DATE` or CHAR(10), that is as `char...[11]`. The date value is always passed to the routine in ISO format:

  `yyyy-mm-dd`

  Example:

  ```
  char     arg1[11];      /* example for DATE       */
  char    *result;        /* also acceptable */
  ```

  Note: For DATE, TIME and TIMESTAMP return values, DB2 demands the characters be in the defined form, and if this is not the case the value could be misinterpreted by DB2 (For example, 2001-04-03 will be interpreted as April 3 even if March 4 is intended) or will cause an error (SQLCODE -493, SQLSTATE 22007).

- TIME

  Valid. Represent in C same as `SQLUDF_TIME` or CHAR(8), that is, as `char...[9]`. The time value is always passed to the routine in ISO format:

  `hh.mm.ss`

Example:

```
char    *arg;           /* example for DATE          */
char     result[9];     /* also acceptable */
```

- TIMESTAMP

  Valid. Represent in C as `SQLUDF_STAMP` or 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 acceptable */
  ```

- GRAPHIC(n)

  Valid. Represent in C as `SQLUDF_GRAPH` or `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.

  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 routine 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. For a RETURNS GRAPHIC(*n*) value or an output parameter of a stored procedure, DB2 looks for an embedded GRAPHIC null CHAR, and if it finds it, pads the value out to *n* with GRAPHIC blank characters.

  When defining graphic routine parameters, consider using VARGRAPHIC rather than GRAPHIC as DB2 does not promote VARGRAPHIC arguments to GRAPHIC. For example, suppose you define a routine 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 might not understand the reason for this message. In the preceding 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 acceptable */
  ```

- VARGRAPHIC(n)

Valid. Represent in C as `SQLUDF_GRAPH` or `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.

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 routine 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. For a RETURNS VARGRAPHIC(*n*) value or an output parameter of a stored procedure, the routine 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 acceptable  */
```

- LONG VARGRAPHIC

  Valid. Represent in C as `SQLUDF_LONGVARG` or a structure:

```
struct sqludf_vg
{
   unsigned short length;         /* length of data */
   sqldbchar      data[1];        /* first char of data */
};
```

Note that in the preceding 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.

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 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 routine for parameters using the structure variable `length`. Data passed from DB2 to a routine 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. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine 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   */
```

- BLOB(n) and CLOB(n)

  Valid. Represent in C as `SQLUDF_BLOB`, `SQLUDF_CLOB`, or 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 provides a way to use array logic.

  These are not represented as C null-terminated strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed back to the routine, is the length of the buffer. What the routine 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 `SQLUDF_DBCLOB` or a structure:

  ```
  struct sqludf_lob
  {
      sqluint32 length;      /* length in graphic characters */
      sqldbchar data[1];         /* first byte of lob */
  };
  ```

  Note that in the preceding 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.

  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 provides a way to use array logic.

  These are not represented as null-terminated graphic strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. Data passed from DB2 to a routine 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. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine 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

  Valid for parameters and results of UDFs and methods. It may only be used to modify LOB types or any distinct type that is based on a LOB type. Represent in C as `SQLUDF_LOCATOR` or a four byte integer.

  The locator value can be assigned to any locator host variable with a compatible type and then be used in an SQL statement. This means that locator variables are only useful in UDFs and methods defined with an SQL access indicator of CONTAINS SQL or higher. For compatibility with existing UDFs and methods, the locator APIs are still supported for NOT FENCED NO SQL UDFs. Use of these APIs is not encouraged for new functions.

  Example:

  ```
       sqludf_locator       *arg1;  /* locator argument */
       sqludf_locator       *result; /* locator result */


  EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS CLOB LOCATOR arg_loc;
     SQL TYPE IS CLOB LOCATOR res_loc;
  EXEC SQL END DECLARE SECTION;

  /* Extract some characters from the middle */
  /* of the argument and return them         */
  *arg_loc = arg1;
  EXEC SQL VALUES SUBSTR(arg_loc, 10, 20) INTO :res_loc;
  *result = res_loc;
  ```

- Structured Types

  Valid for parameters and results of UDFs and methods where an appropriate transform function exists. Structured type parameters will be passed to the function or method in the result type of the FROM SQL transform function. Structured type results will be passed in the parameter type of the TO SQL transform function.

- DATALINK

  Valid. Represent in C as `SQLUDF_DATALINK` or a structure similar to the following from the sqludf.h include file:

  ```
  struct sqludf_datalink {
    sqluint32 version;
    char      linktype[4];
    sqluint32 url_length;
    sqluint32 comment_length;
    char      reserve2[8];
    char      url_plus_comment[230];
  }
  ```

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Graphic Host Variables in C/C++ Routines" on page 115
- "Include File for C/C++ Routines (sqludf.h)" on page 102
- "C/C++ Routines" on page 97

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in C and C++" in the *Application Development Guide: Programming Client Applications*
- "Supported SQL Data Types in C/C++" on page 103

Graphic Host Variables in C/C++ Routines

Any routine 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 routine 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. If you do not use WCHARTYPE NOCONVERT, it is still possible for you to manipulate graphic data in `wchar_t` format in a routine; however, you must perform the input and output conversions manually.

CONVERT can be used in FENCED routines, and it will affect the graphic data in SQL statements within the routine, but not data passed through the routine's parameters. NOT FENCED routines must be built using the NOCONVERT option.

In summary, graphic data passed to or returned from a routine through its input or output parameters is in DBCS format, regardless of how it was precompiled with the WCHARTYPE option.

Related concepts:

- "WCHARTYPE Precompiler Option in C and C++" in the *Application Development Guide: Programming Client Applications*
- "WCHARTYPE CONVERT Precompile Option" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

C++ Type Decoration

The names of C++ functions can be overloaded. Two C++ functions with the same name can coexist if they have different arguments, for example:

```
int func( int i )
```

and

```
int func( char c )
```

C++ compilers type-decorate or 'mangle' function names by default. This means that argument type names are appended to their function names to resolve them, as in `func__Fi` and `func__Fc` for the two earlier examples. The mangled names will be different on each platform, so code that explicitly uses a mangled name is not portable.

On Windows® operating systems, the type-decorated function name can be determined from the `.obj` (object) file.

With the Microsoft® Visual C++ compiler on Windows, you can use the `dumpbin` command to determine the type-decorated function name from the `.obj` (object) file, as follows:

```
dumpbin /symbols myprog.obj
```

where `myprog.obj` is your program object file.

On UNIX® platforms, the type-decorated function name can be determined from the `.o` (object) file, or from the shared library, using the `nm` command.

This command can produce considerable output, so it is suggested that you pipe the output through `grep` to look for the right line, as follows:

```
nm myprog.o | grep myfunc
```

where `myprog.o` is your program object file, and `myfunc` is the function in the program source file.

The output produced by all of these commands includes a line with the mangled function name. On UNIX, for example, this line is similar to the following:

```
myfunc__FPlT1PsT3PcN35|     3792|unamex|         | ...
```

Once you have obtained the mangled function name from one of the preceding commands, you can use it in the appropriate command. This is demonstrated later in this section using the mangled function name obtained from the preceding UNIX example. A mangled function name obtained on Windows would be used in the same way.

When registering a routine with the CREATE statement, the EXTERNAL NAME clause must specify the mangled function name. For example:

```
CREATE FUNCTION myfunco(...) RETURNS...
       ...
       EXTERNAL NAME '/whatever/path/myprog!myfunc__FPlT1PsT3PcN35'
       ...
```

If your routine library does not contain overloaded C++ function names, you have the option of using `extern "C"` to force the compiler to not type-decorate function names. (Note that you can always overload the SQL function names given to UDFs, because DB2® resolves what library function to invoke based on the name and the parameters it takes.)

```
#include <string.h>
#include <stdlib.h>
#include "sqludf.h"

/*---------------------------------------------------------------------*/
/* function fold: output = input string is folded at point indicated   */
/*                         by the second argument.                     */
/*        inputs: CLOB,                 input string                   */
/*                LONG                  position to fold on            */
/*        output: CLOB                  folded string                  */
/*---------------------------------------------------------------------*/
extern "C" void fold(
      SQLUDF_CLOB      *in1,                    /* input CLOB to fold */
   ...
   ...
}
/* end of UDF: fold */

/*---------------------------------------------------------------------*/
/* function find_vowel:                                                */
/*             returns the position of the first vowel.                */
/*             returns error if no vowel.                              */
/*             defined as NOT NULL CALL                                */
/*        inputs: VARCHAR(500)                                         */
/*        output: INTEGER                                              */
/*---------------------------------------------------------------------*/
extern "C" void findvwl(
      SQLUDF_VARCHAR    *in,                    /* input smallint */
   ...
   ...
}
/* end of UDF: findvwl */
```

In this example, the UDFs `fold` and `findvwl` are not type-decorated by the compiler, and should be registered in the CREATE FUNCTION statement using their plain names. Similarly, if a C++ stored procedure or method is coded with `extern "C"`, its undecorated function name would be used in the CREATE statement.

Related concepts:

- "C/C++ Routines" on page 97

Java Routines

The following sections describe how to write Java routines.

Java Routines

When developing routines in Java, it is strongly recommended that you register them using the PARAMETER STYLE JAVA clause in the CREATE statement. With PARAMETER STYLE JAVA, a routine will use a parameter passing convention that conforms to the Java™ language and SQLj Routines specification.

There are some UDF and method features that cannot be implemented with PARAMETER STYLE JAVA. These are as follows:

- table functions
- scratchpads
- access to the DBINFO structure
- the ability to make a FINAL CALL (and a separate first call) to the function or method

If you need to implement the above features in a UDF or method you can either write your routine in C, or write it in Java, using PARAMETER STYLE DB2GENERAL. Aside from these specific cases, all mentions of Java routines in this documentation will assume the use of PARAMETER STYLE JAVA.

Java UDFs and methods:

The signature of PARAMETER STYLE JAVA UDFs and methods follows this format:

```
public static return-type method-name ( SQL-arguments ) throws SQLException
```

return-type
: The data type of the value to be returned by the scalar routine. Inside the routine, the return value is passed back to the invoker through a return statement.

method-name
: Name of the method. During routine registration, this value is specified with the class name in the EXTERNAL NAME clause of the routine's CREATE statement.

SQL-arguments
: Corresponds to the list of input parameters in the routine's CREATE statement.

The following is an example of a Java UDF that returns the product of its two input arguments:

```
public static double product( double in1, double in2 ) throws SQLException
{
  return in1 * in2;
}
```

The corresponding CREATE FUNCTION statement for this UDF is as follows:

```
CREATE FUNCTION product( double in1, double in2 )
  RETURNS double
  LANGUAGE java
  PARAMETER STYLE java
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'myjar:udfclass.product'
```

The preceding statement assumes that the method is in a class called `udfclass` which lives in a JAR file that has been cataloged to the database with the Jar ID `myjar`

Java stored procedures:

The signature of PARAMETER STYLE JAVA stored procedures follows this format:

```
public static void method-name ( SQL-arguments, ResultSet[] result-set-array )
                                throws SQLException
```

method-name
: Name of the method. During routine registration, this value is specified with the class name in the EXTERNAL NAME clause of the CREATE PROCEDURE statement.

SQL-arguments
: Corresponds to the list of input parameters in the CREATE PROCEDURE statement. OUT or INOUT mode parameters are passed as single-element arrays. For each result set that is specified in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement, a single-element array of type ResultSet is appended to the parameter list.

result-set-array
: Name of the array of ResultSet objects. For every result set declared in the DYNAMIC RESULT SETS parameter of the CREATE PROCEDURE statement, a parameter of type ResultSet[] must be declared in the Java method signature.

The following is an example of a Java stored procedure that accepts an input parameter, and then returns an output parameter and a result set:

```
public static void javastp( int inparm, int[] outparm, ResultSet[] rs )
                   throws SQLException
{
  Connection con = DriverManager.getConnection( "jdbc:default:connection" );
  PreparedStatement stmt = null;
  String sql = SELECT value FROM table01 WHERE index = ?";
  //Prepare the query with the value of index
  stmt = con.prepareStatement( sql );
  stmt.setInt( 1, inparm );

  //Execute query and set output parm
  rs[0] = stmt.executeQuery();
  outparm[0] = inparm + 1;

  //Close open resources
  if (stmt != null) stmt.close();
  if (con != null) con.close();

  return;
}
```

The corresponding CREATE PROCEDURE statement for this stored procedure is as follows:

```
CREATE PROCEDURE javaproc( IN in1 INT, OUT out1 INT )
  LANGUAGE java
  PARAMETER STYLE java
  DYNAMIC RESULT SETS 1
  FENCED THREADSAFE
  EXTERNAL NAME 'myjar:stpclass.javastp'
```

The preceding statement assumes that the method is in a class called `stpclass`, which exists in a JAR file that has been cataloged to the database with the Jar ID `myjar`

Notes:

1. PARAMETER STYLE JAVA routines use exceptions to pass error data back to the invoker. For complete information, including the exception call stack, refer to db2diag.log. Other than this detail, there are no other special considerations for invoking PARAMETER STYLE JAVA routines.
2. JNI calls are not supported in Java routines. However, it is possible to invoke C functionality from Java routines by nesting an invocation of a C routine. This involves moving the desired C functionality into a routine, registering it, and invoking it from within the Java routine.

Related concepts:

- "DB2GENERAL Routines" on page 303
- "Table Function Execution Model for Java" on page 57

Related tasks:

- "Debugging Stored Procedures in Java" on page 125
- "Building JDBC Routines" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ Routines" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Scalar) statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in Java" on page 123
- "JAR File Administration on the Database Server" on page 122
- "JDBC Samples" in the *Application Development Guide: Building and Running Applications*
- "SQLJ Samples" in the *Application Development Guide: Building and Running Applications*
- "CREATE PROCEDURE (External) statement" in the *SQL Reference, Volume 2*

Related samples:

- "SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)"
- "UDFjsrv.java -- Provide UDFs to be called by UDFjcli.java (JDBC)"
- "UDFsqlsv.java -- Provide UDFs to be called by UDFsqlcl.java (JDBC)"
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)"
- "SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)"
- "UDFjsrv.java -- Provide UDFs to be called by UDFjcli.sqlj (SQLj)"
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.sqlj (SQLj)"

JAR File Administration on the Database Server

The Java class files that you use to implement a routine must reside in either a JAR file you have installed in the database, or in the correct CLASSPATH for your operating system. The DB2 classloader searches the classes and JAR files in the CLASSPATH and will pick up the first class it encounters with the specified name.

To install, replace, or remove a JAR file in a DB2 instance, use the stored procedures provided with DB2:

Install

```
sqlj.install_jar( jar-url, jar-id )
```

Replace

```
sqlj.replace_jar( jar-url, jar-id )
```

Remove

```
sqlj.remove_jar( jar-id )
```

- *jar-url*: Specifies the URL containing the JAR file to be installed or replaced. The only URL scheme supported is 'file:'.
- *jar-id*: A unique string identifier, up to 128 bytes in length. It specifies the JAR identifier in the database associated with the *jar-url* file.

Note: When invoked from applications, the stored procedures sqlj.install_jar and sqlj.remove_jar have an additional parameter. It is an integer value that dictates the use of the deployment descriptor in the specified JAR file. At present, the deployment parameter is not supported, and any invocation specifying a nonzero value will be rejected.

Following are a series of examples of how to use the preceding JAR file management stored procedures.

To register a JAR located in the path /home/bob/bobsjar.jar with the database instance as MYJAR:

```
CALL sqlj.install_jar( 'file:/home/bob/bobsjar.jar', 'MYJAR' )
```

Subsequent SQL commands that use the `bobsjar.jar` file refer to it with the name `MYJAR`.

To replace MYJAR with a different JAR containing some updated classes:

```
CALL sqlj.replace_jar( 'file:/home/bob/bobsnewjar.jar', 'MYJAR' )
```

To remove MYJAR from the database catalogs:

```
CALL sqlj.remove_jar( 'MYJAR' )
```

Note: On Windows 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.

Related concepts:
- "Where to Put Java Classes" in the *Application Development Guide: Programming Client Applications*
- "Java Routines" on page 118
- "Library and Class Management Considerations for Developing Routines" on page 23

Supported SQL Data Types in Java

The following table shows the Java equivalent of each SQL data type, based on the JDBC specification for data type mappings. 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 procedures and UDFs.

Note: There is no host variable support for the DATALINK data type in any of the programming languages supported by DB2.

Table 5. 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)	java.lang.String	Fixed-length character string of length *n* where *n* is from 1 to 254
CHAR(*n*) FOR BIT DATA	byte[]	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	java.lang.String	Variable-length character string
VARCHAR(*n*) FOR BIT DATA	byte[]	Variable-length character string
LONG VARCHAR (456 or 457)	java.lang.String	Long variable-length character string
LONG VARCHAR FOR BIT DATA	byte[]	Long variable-length character string
BLOB(*n*) (404 or 405)	java.sql.Blob	Large object variable-length binary string
CLOB(*n*) (408 or 409)	java.sql.Clob	Large object variable-length character string
DBCLOB(*n*) (412 or 413)	java.sql.Clob	Large object variable-length double-byte character string
DATE (384 or 385)	java.sql.Date	10-byte character string

Table 5. SQL Data Types Mapped to Java Declarations (continued)

SQL Column Type	Java Data Type	SQL Column Type Description
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	26-byte character string

Debugging Stored Procedures in Java

The following sections describe how to debug Java stored procedures.

Debugging Stored Procedures in Java

DB2 provides the capability to interactively debug a stored procedure written in JDBC when it executes on an AIX, Linux, Solaris, Windows NT, or Windows 2000 server. The easiest way to invoke debugging is through the DB2 Development Center.

Procedure:

To debug stored procedures in Java:

1. Prepare to debug.
2. Populate the debug table.
3. Invoke the debugger.

Related tasks:

- "Preparing to Debug Java Stored Procedures" on page 125
- "Populating the Debug Table" on page 127
- "Invoking the Debug Program" on page 128
- "Debugging Routines" on page 31
- "Debugging : Development Center help" in the *Help: Development Center*

Preparing to Debug Java Stored Procedures

When preparing to interactively debug a Java stored procedure, you work with the stored procedure, the client, and the server.

Procedure:

To prepare to debug Java stored procedures:

1. Compile the stored procedure in debug mode according to your JDK documentation.
2. Prepare the server.

a. If the source code is 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 described in the JAR File Administration on the Database Server topic.

b. Enter 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 that contains the source code for the stored procedure.

4. Create the debug table.

   If you do not use the Development Center to invoke the debug program, create the debug table with the following command:

   ```
   db2 -tf sqllib/misc/db2debug.ddl
   ```

   Note: In partitioned database environments, the default database partition group is IBMDEFAULTGROUP for the USERSPACE1 table space, and it spans all the database partitions. To improve the performance of debugging stored procedures in a partitioned database environment, you should have a single coordinator partition where debugging will occur, and define a database partition group that only contains that database partition.

5. Start the debug program on the client.

   From the stored procedure client, start the debug program with the following command:

   ```
   db2dbugd -qport=portno
   ```

   where *portno* is an unused TCP/IP port number. If you do not supply a value, the debug program uses 8000 as the default port number. On Windows operating systems, you can also click the debug program shortcut located in the DB2 folder to start the debug program with the default port number.

You are now ready to populate the debug table.

Related concepts:

- "Where to Put Java Classes" in the *Application Development Guide: Programming Client Applications*

Related tasks:

- "Populating the Debug Table" on page 127
- "Invoking the Debug Program" on page 128
- "Debugging Routines" on page 31

Related reference:

- "Java Debug Table DB2DBG.ROUTINE_DEBUG" on page 128
- "JAR File Administration on the Database Server" on page 122

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 on the table 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. The rest of this section assumes that you are populating that table through the user view.

Procedure:

If you use the Development Center to invoke debugging, you can use the debug program 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 192.168.111.222, 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', '192.168.111.222')
```

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. However, old debug information can cause some confusion if a stored procedure is recreated. If you want to keep the debug table synchronized with the DB2 catalog, you must delete the debug information manually.

You are now ready to invoke the debug program.

Related tasks:

- "Preparing to Debug Java Stored Procedures" on page 125
- "Invoking the Debug Program" on page 128
- "Debugging Routines" on page 31

Related reference:

- "Java Debug Table DB2DBG.ROUTINE_DEBUG" on page 128

Invoking the Debug Program

In the debug program, you can step through the source code, display variables, and set breakpoints in the source code.

Procedure:

After you have prepared to debug and populated the debug table, call the stored procedure that you want to debug. This action invokes the debug program on the client using the IP address that you specified in the debug table.

Related tasks:

- "Preparing to Debug Java Stored Procedures" on page 125
- "Populating the Debug Table" on page 127
- "Debugging Routines" on page 31
- "Debugging : Development Center help" in the *Help: Development Center*

Related reference:

- "Java Debug Table DB2DBG.ROUTINE_DEBUG" on page 128

Java Debug Table DB2DBG.ROUTINE_DEBUG

Whether you create the debug table manually or through the Development Center, the debug table is named DB2DBG.ROUTINE_DEBUG and has the following definition:

Table 6. 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’ (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. • **N** - disables debugging for the stored procedure. This is the default.
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.			

The DB2DBG.ROUTINE_DEBUG_USER view limits the access to this table only to rows belonging to the user connected to the database.

Related tasks:

- “Debugging Stored Procedures in Java” on page 125
- “Preparing to Debug Java Stored Procedures” on page 125
- “Populating the Debug Table” on page 127
- “Invoking the Debug Program” on page 128
- “Debugging Routines” on page 31

OLE Automation Routines

The following sections describe how to write OLE automation routines.

OLE Automation Routine Design

Object Linking and Embedding (OLE) 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 is the foundation of OLE. For OLE automation routines, DB2® acts as an OLE automation controller. Through this mechanism, DB2 can invoke methods of OLE automation objects as external routines.

Note that all OLE automation topics assume that you are familiar with OLE automation terms and concepts. 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.

Related concepts:

- "Object Instance and Scratchpad Considerations and OLE Routines" on page 132
- "OLE Automation Routines in BASIC and C++" on page 134

Related tasks:

- "Creating and Registering OLE Automation Routines" on page 130

Related reference:

- "Supported SQL Data Types in OLE Automation" on page 133

Creating and Registering OLE Automation Routines

OLE automation routines 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 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).

Procedure:

To register OLE automation routines:

After you code an OLE automation object, you need to register the methods of the object as routines using the CREATE statement. Registering OLE automation routines is very similar to registering C or C++ routines, but you must use the following options:

- LANGUAGE OLE
- FENCED NOT THREADSAFE, since OLE automation routines must run in FENCED mode, but cannot be run as THREADSAFE.

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
  NOT THREADSAFE
  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 routines written in C or C++. An implementation of the previous 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)
```

Related concepts:

- "Object Linking and Embedding (OLE) Automation with Visual Basic" in the *Application Development Guide: Building and Running Applications*

- “Object Linking and Embedding (OLE) Automation with Visual C++” in the *Application Development Guide: Building and Running Applications*
- “OLE Automation Routine Design” on page 130
- “Object Instance and Scratchpad Considerations and OLE Routines” on page 132
- “OLE Automation Routines in BASIC and C++” on page 134

Related reference:
- “CREATE TYPE (Structured) statement” in the *SQL Reference, Volume 2*
- “CREATE FUNCTION (External Scalar) statement” in the *SQL Reference, Volume 2*
- “Object Linking and Embedding (OLE) Samples” in the *Application Development Guide: Building and Running Applications*
- “CREATE PROCEDURE (External) statement” in the *SQL Reference, Volume 2*
- “Supported SQL Data Types in OLE Automation” on page 133

Object Instance and Scratchpad Considerations and OLE Routines

OLE automation UDFs and methods (methods of OLE automation objects) are applied on instances of OLE automation objects. DB2® creates an object instance for each UDF or method reference in an SQL statement. An object instance can be reused for subsequent method invocations of the UDF or method 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 CREATE 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.

Using the scratchpad allows a method to maintain state information in instance variables of the object, across function or method invocations. It also increases performance as an object instance is only created once and then reused for subsequent invocations.

Related concepts:
- “OLE Automation Routine Design” on page 130
- “OLE Automation Routines in BASIC and C++” on page 134

Related tasks:
- “Creating and Registering OLE Automation Routines” on page 130

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Scalar) statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE (External) statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in OLE Automation" on page 133

Supported SQL Data Types in OLE Automation

DB2 handles type conversion between SQL types and OLE automation types. The following table summarizes the supported data types and how they are mapped.

Table 7. 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	

Table 7. Mapping of SQL and OLE Automation Datatypes (continued)

SQL Type	OLE Automation Type	OLE Automation Type Description
CHAR(*n*)	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*)	SAFEARRAY[unsigned char]	
LONG VARCHAR	SAFEARRAY[unsigned char]	
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	
BLOB(*n*)	SAFEARRAY[unsigned char]	

Data passed between DB2 and OLE automation routines is passed as call by reference. SQL types such as BIGINT, DECIMAL, DATALINK, 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 scheme. (UCS-2 is also known as Unicode, IBM code page 13488). Upon return, the data is converted back to the database code page from UCS-2. These conversions occur regardless of the database code page. If these code page conversion tables are not installed, you receive SQLCODE -332 (SQLSTATE 57017).

Related concepts:

- "OLE Automation Routine Design" on page 130
- "Object Instance and Scratchpad Considerations and OLE Routines" on page 132
- "OLE Automation Routines in BASIC and C++" on page 134

Related tasks:

- "Creating and Registering OLE Automation Routines" on page 130

OLE Automation Routines in BASIC and C++

You can implement OLE automation routines in any language. This section shows you how to implement OLE automation routines using BASIC or C++ as two sample languages. The following table shows the mapping of OLE automation types to data types in BASIC and C++.

Table 8. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types

SQL Type	OLE Automation Type	BASIC Type	C++ Type
SMALLINT	short	Integer	short
INTEGER	long	Long	long
REAL	float	Single	float
FLOAT or DOUBLE	double	Double	double
DATE, TIME, TIMESTAMP	DATE	Date	DATE
CHAR(*n*)	BSTR	String	BSTR
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
VARCHAR(*n*)	BSTR	String	BSTR
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
LONG VARCHAR	BSTR	String	BSTR
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
BLOB(*n*)	BSTR	String	BSTR
BLOB(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
GRAPHIC(*n*), VARGRAPHIC(*n*), LONG GRAPHIC, DBCLOB(*n*)	BSTR	String	BSTR

OLE Automation in BASIC:

To implement OLE automation routines 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 OLE automation UDF, `bcounter`, 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)
```

OLE Automation in C++:

The C++ declaration of the OLE automation UDF, `increment`, 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 you to specify whether a parameter is an input (in), output (out), or input/output (in,out) parameter. For an OLE automation routine, the routine input parameters and input indicators are specified as [in] parameters, and routine output parameters and output indicators as [out] parameters. For the routine trailing arguments, `sqlstate` is an [out] parameter, `fname` and `fspecname` are [in] parameters, `scratchpad` is an [in,out] parameter, and `calltype` is an [in] parameter.

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 called routine frees a BSTR passed in as a by-reference parameter before assigning the parameter a new value. The same rule applies for one-dimensional byte arrays that are received by the called routine as `SAFEARRAY**`. This rule means the following for DB2® and OLE automation routines:

- [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 invoked routine and is freed by DB2.
- [in,out] parameters: DB2 initially allocates [in,out] parameters. They can be freed and re-allocated by the invoked routine. 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++ routine 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.

Related concepts:

- "Object Linking and Embedding (OLE) Automation with Visual Basic" in the *Application Development Guide: Building and Running Applications*
- "Object Linking and Embedding (OLE) Automation with Visual C++" in the *Application Development Guide: Building and Running Applications*
- "OLE Automation Routine Design" on page 130
- "Object Instance and Scratchpad Considerations and OLE Routines" on page 132

Related tasks:

- "Creating and Registering OLE Automation Routines" on page 130

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "Object Linking and Embedding (OLE) Samples" in the *Application Development Guide: Building and Running Applications*
- "Supported SQL Data Types in OLE Automation" on page 133

OLE DB User-Defined Table Functions

The following sections describe how to write OLE DB table functions.

OLE DB User-Defined 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 as LANGUAGE 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 the OLE DB specification, see the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Related concepts:

- "Object Linking and Embedding Database (OLE DB) Table Functions" in the *Application Development Guide: Building and Running Applications*
- "Fully Qualified Rowset Names" on page 142

Related tasks:

- "Creating an OLE DB Table UDF" on page 139

Related reference:

- "CREATE FUNCTION (OLE DB External Table) statement" in the *SQL Reference, Volume 2*
- "Object Linking and Embedding Database (OLE DB) Table Function Samples" in the *Application Development Guide: Building and Running Applications*
- "Supported SQL Data Types in OLE DB" on page 143

Creating an OLE DB Table UDF

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 a *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"`.

The EXTERNAL NAME clause can take either of the following forms:

```
   'server!rowset'
or
   '!rowset!connectstring'
```

where:

server identifies a server registered with the 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`, 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 that 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;
```

Related concepts:

- "Fully Qualified Rowset Names" on page 142
- "OLE DB User-Defined Table Functions" on page 138
- "Create the wrapper" in the *Federated Systems Guide*

Related tasks:

- "Adding OLE DB data sources to a federated server" in the *Federated Systems Guide*
- "Setting up the server to access OLE DB data sources" in the *Federated Systems Guide*

Related reference:

- "CREATE NICKNAME statement" in the *SQL Reference, Volume 2*
- "CREATE SERVER statement" in the *SQL Reference, Volume 2*
- "CREATE WRAPPER statement" in the *SQL Reference, Volume 2*

- "CREATE FUNCTION (OLE DB External Table) statement" in the *SQL Reference, Volume 2*
- "Object Linking and Embedding Database (OLE DB) Table Function Samples" in the *Application Development Guide: Building and Running Applications*
- "Supported SQL Data Types in OLE DB" on page 143

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.

Related concepts:

- "OLE DB User-Defined Table Functions" on page 138

Related reference:

- "CREATE FUNCTION (OLE DB External Table) statement" in the *SQL Reference, Volume 2*

Supported SQL Data Types in OLE DB

The following table shows how DB2 data types map to the OLE DB data types as 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 9. 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

Table 9. Mapping DB2 Data Types to OLE DB (continued)

DB2 Data Type	OLE DB Data Type
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 can 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 can 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 can get converted to OLE DB data type DBTYPE_BYTES, which maps to DB2 data type CHAR(12) FOR BIT DATA.

Related concepts:

- "OLE DB User-Defined Table Functions" on page 138

Related tasks:

- "Creating an OLE DB Table UDF" on page 139

Chapter 5. Invoking/Calling Routines

Invoking Routines

Once a routine is written, compiled, linked, and registered with the database, it can be invoked.

Prerequisites:

In order to invoke the routine, it must be registered with the database, and the library or class file must be installed in the correct location. If the library is not present in the location specified by the CREATE statement, error SQL0444 will occur.

Procedure:

Each type of routine is invoked differently, as is reflective of their varying natures.

To invoke a particular type of routine:
- Invoke a procedure.
- Invoke a scalar UDF or method from an SQL expression.
- Invoke a user-defined table function from the FROM clause of a SELECT statement.

Related concepts:
- "Routines (Stored Procedures, UDFs, Methods)" on page 3

Related tasks:
- "Invoking Stored Procedures" on page 146
- "Invoking UDFs" on page 148
- "Invoking User-Defined Table Functions" on page 149
- "Writing Routines" on page 29

- “Registering Routines” on page 27

Invoking Stored Procedures

Once a stored procedure is written and registered with the database, you can invoke it by using the CALL statement. The CALL statement can pass parameters to the stored procedure and receive parameters returned from the stored procedure. Any result sets returned by the stored procedure can be processed once the stored procedure has finished running.

You can write the client application that invokes the stored procedure in a different language than the one used to write the stored procedure. In addition, the client application that invokes the stored procedure can be executed on a different platform than the one where the stored procedure resides.

Prerequisites:

In order to invoke the stored procedure, it must be registered with the database.

For the list of privileges required to invoke stored procedures, see the CALL statement.

Procedure:

To invoke a stored procedure from an application or routine:

1. Declare, allocate, and initialize storage for the optional data structures and host variables or parameter markers.

   This involves the following steps:

   - Assign a host variable or parameter marker to each parameter of the stored procedure.
   - Initialize the host variables or parameter markers that correspond to IN or INOUT parameters.
   - Invoke the stored procedure.
   - Process the data in the OUT and INOUT host variables or parameter markers, as well as any result sets.
2. Connect to a database by executing the CONNECT TO statement, or by doing an implicit connect.
3. Invoke the stored procedure through the SQL CALL statement.
4. Issue 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 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. Stored procedures cannot issue COMMIT or ROLLBACK statements if the stored procedure was invoked from an application that established a type 2 connection to the database.

5. Disconnect from the database.

Note: You can code SQL statements at any point between steps 2 and 5.

Related concepts:

- "Routines: Stored Procedures" on page 7

Related tasks:

- "Calling Stored Procedures with the CALL Statement" in the *Application Development Guide: Building and Running Applications*
- "Invoking Routines" on page 145

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*
- "COMMIT statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "ROLLBACK statement" in the *SQL Reference, Volume 2*

Related samples:

- "spclient.c -- Call various stored procedures (CLI)"
- "spclient.sqc -- Call various stored procedures (C)"
- "spclient.sqC -- Call various stored procedures (C++)"
- "SpClient.java -- Call a variety of types of stored procedures from SpServer.java (JDBC)"
- "SpClient.sqlj -- Call a variety of types of stored procedures from SpServer.sqlj (SQLj)"

Invoking UDFs

Once the UDF is written and registered with the database, you can invoke it within an SQL statement wherever an expression is valid.

Restrictions:

For restrictions on invoking UDFs, see the CREATE FUNCTION topics in the related links.

Prerequisites:

In order to invoke the UDF, it must be registered with the database.

Procedure:

To invoke a scalar UDF, include it in an SQL statement where it is to process a set of input values. A scalar UDF can be invoked anywhere an expression is valid.

For example, suppose that you have defined a UDF called TOTAL_SAL that adds the base salary and bonus together for each employee row in the EMPLOYEE table.

```
CREATE FUNCTION TOTAL_SAL
  (SALARY DECIMAL(9,2), BONUS DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SALARY+BONUS
```

The following is a SELECT statement that makes use of TOTAL_SAL:

```
SELECT LASTNAME, TOTAL_SAL(SALARY, BONUS) AS TOTAL
  FROM EMPLOYEE
```

Related concepts:

- "References to Functions" on page 156
- "Routine Names and Paths" on page 154

Related tasks:

- "Invoking User-Defined Table Functions" on page 149

Related reference:

- "SELECT statement" in the *SQL Reference, Volume 2*

- "CREATE FUNCTION (SQL Scalar, Table or Row) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Scalar) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (Sourced or Template) statement" in the *SQL Reference, Volume 2*

Related samples:

- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfcli.sqC -- Call a variety of types of user-defined functions (C++)"
- "udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)"
- "UDFcli.java -- Call the UDFs in UDFsrv.java (JDBC)"
- "UDFjcli.java -- Call the UDFs in UDFjsrv.java (JDBC)"
- "UDFcli.sqlj -- Call the UDFs in UDFsrv.java (SQLj)"
- "UDFjcli.sqlj -- Call the UDFs in UDFjsrv.java (SQLj)"

Invoking User-Defined Table Functions

Once the user-defined table function is written and registered with the database, you can invoke it in the FROM clause of a SELECT statement.

Restrictions:

For restrictions on invoking user-defined table functions, see the CREATE FUNCTION topics in the related links.

Prerequisites:

In order to invoke the user-defined table function, it must be registered with the database.

Procedure:

To invoke a user-defined table function, include it in the FROM clause of a SELECT statement where it is to process a set of input values.

For example, the following CREATE FUNCTION statement defines a table function that returns the employees in a specified department number.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO VARCHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
    SELECT EMPNO, LASTNAME, FIRSTNME FROM EMPLOYEE
      WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

The following is a SELECT statement that makes use of DEPTEMPLOYEES:

```
SELECT EMPNO, LASTNAME, FIRSTNAME FROM TABLE(DEPTEMPLOYEES('A00')) AS D
```

Related concepts:
- "References to Functions" on page 156
- "Routine Names and Paths" on page 154

Related tasks:
- "Invoking Routines" on page 145
- "Invoking UDFs" on page 148

Related reference:
- "CREATE FUNCTION (OLE DB External Table) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (SQL Scalar, Table or Row) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Table) statement" in the *SQL Reference, Volume 2*

Related samples:
- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfcli.sqC -- Call a variety of types of user-defined functions (C++)"
- "udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)"
- "UDFcli.java -- Call the UDFs in UDFsrv.java (JDBC)"
- "UDFjcli.java -- Call the UDFs in UDFjsrv.java (JDBC)"
- "UDFcli.sqlj -- Call the UDFs in UDFsrv.java (SQLj)"
- "UDFjcli.sqlj -- Call the UDFs in UDFjsrv.java (SQLj)"

Routine Nesting

In the context of routines, *nesting* refers to the situation where one routine invokes another. That is to say, the SQL issued by one routine can reference another routine, which could issue SQL that again references another routine, and so on. If, in this nesting, we encounter a routine that is already in the current ″stack″, we are said to be in a *recursive* nesting situation.

You can use nesting and recursion in your DB2® routines under the following restrictions:

16 levels of nesting
You can nest routine invocations up to 16 levels deep. Consider a scenario in which routine A calls routine B, and routine B calls routine C. In this example, the execution of routine C is at nesting level 3. A further thirteen levels of nesting are possible.

Restrictions
A routine cannot call a target routine that is cataloged with a higher SQL data access level. For example, a UDF created with the CONTAINS SQL clause can call stored procedures created with either the CONTAINS SQL clause or the NO SQL clause. However, this routine cannot call stored procedures created with either the READS SQL DATA clause or the MODIFIES SQL DATA clause (SQLCODE -577, SQLSTATE 38002). This is because the invoker's SQL level does not allow any read or modify operations to occur (this is inherited by the routine being invoked).

Another limitation when nesting routines is that access to tables is restricted to prevent conflicting read and write operations between routines.

Related concepts:
- "Conflicts When Reading and Writing Tables From Routines" on page 33
- "Security Considerations for Routines" on page 20

Invoking 32-bit Routines on a 64-bit Database Server

It is possible to invoke 32-bit routines on a 64-bit database server. The first time a 32--bit routine is invoked in such an environment, there is a performance degradation. Subsequent invocations of the 32-bit stored procedure will perform the same as an equivalent 64-bit routine.

For Java stored procedures, a 32-bit JVM can function on a 64-bit database server. For 32-bit Java routines using this JVM, there is no additional performance overhead. A comparable 64-bit routine using a 64-bit JVM will run no faster. However, a 32-bit Java routine running on a 64-bit database server will not scale well due to the restriction of needing to run in FENCED NOT THREADSAFE mode. Because of this, every invocation of such a routine will require its own JVM.

Restrictions:

32-bit routines must be registered as FENCED and NOT THREADSAFE in order to work in a 64-bit instance.

It is not possible to invoke 32-bit routines on a Linux/IA-64 database server.

Procedure:

To run existing 32-bit routines on a 64-bit server:
1. Copy the routine class or library to the database routines directory:
   - UNIX: sqllib/function
   - Windows: sqllib\function
2. Register the stored procedure with the CREATE PROCEDURE statement.
3. Invoke the stored procedure with the CALL statement.

Related concepts:
- "Java Routines" on page 118

Related tasks:
- "Invoking Routines" on page 145

Code Pages and Routines

Character data is passed to external routines in the code page of the database. Likewise, a character string that is output from the routine is assumed by the database to use that database's code page.

When a client program (using, for example, code page A) invokes a routine that accesses a database using a different code page (for example, code page Z), the following events occur:
1. When an SQL statement is invoked, input character data is converted from the application code page (A) to the one associated with the database (Z). Conversion does not occur for BLOBs or data that will be used as FOR BIT DATA.
2. Once the input data is converted, the database manager does not perform any code page conversions before invoking the routine. All SQL is

performed in the codepage of the database. Therefore, **you must run the routine using the same code page as the database**, in this example, code page Z.

It is strongly recommended that you precompile, compile, and bind the server routine using the same code page as the database. This might not be possible in all cases. For example, you can create a Unicode database in a Windows® environment. However, if the Windows environment does not have the Unicode code page, you would have to precompile, compile, and bind the application that creates the routine in a Windows code page. The routine will work if the application has no special delimiter characters that the precompiler does not understand.

3. When the statement finishes, the database manager converts all output character data from the database code page (Z) back to the application code page (A). If a routine raised an error during the execution of the SQL statement, the SQLSTATE and diagnostic message from the routine will be converted to the application code page (A). Conversion does not occur for BLOBs or for data that was used as FOR BIT DATA.

By using the DBINFO option on the CREATE FUNCTION, CREATE PROCEDURE, and CREATE TYPE statements, the database code page is passed to the routine. Using this information, a routine that is sensitive to the code page can be written to operate in many different code pages.

Related concepts:

- "Character conversion" in the *SQL Reference, Volume 1*
- "Derivation of Code Page Values" in the *Application Development Guide: Programming Client Applications*
- "Active Code Page for Precompilation and Binding" in the *Application Development Guide: Programming Client Applications*
- "Active Code Page for Application Execution" in the *Application Development Guide: Programming Client Applications*
- "Character Conversion Between Different Code Pages" in the *Application Development Guide: Programming Client Applications*
- "When Code Page Conversion Occurs" in the *Application Development Guide: Programming Client Applications*
- "Character Substitutions During Code Page Conversions" in the *Application Development Guide: Programming Client Applications*
- "Supported Code Page Conversions" in the *Application Development Guide: Programming Client Applications*
- "Application Development in Unequal Code Page Situations" in the *Application Development Guide: Programming Client Applications*

Related reference:

- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*
- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*
- “CREATE TYPE (Structured) statement” in the *SQL Reference, Volume 2*
- “CREATE METHOD statement” in the *SQL Reference, Volume 2*
- “Supported territory codes and code pages” in the *Administration Guide: Planning*
- “Conversion tables for code pages 923 and 924” in the *Administration Guide: Planning*

Referencing Routines

The following sections describe how the database server identifies, references, and selects routines when you invoke them.

Routine Names and Paths

The qualified name of a stored procedure or UDF is `schema-name.routine-name`. You can use this qualified name anywhere you refer to a stored procedure or UDF. For example:

```
SANDRA.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
```

However, you may also omit the `schema-name.`, in which case, DB2® will attempt to identify the stored procedure or UDF to which you are referring. For example:

```
BOAT_COMPARE    FOO    SUBSTR    FLOOR
```

The qualified name of a method is `schema-name.type..method-name`.

The concept of *SQL path* is central to DB2’s resolution of *unqualified* references that occur when you do not use the `schema-name`. The SQL path is an ordered list of schema names. It provides a set of schemas for resolving unqualified references to stored procedures, UDFs, and types. In cases where a reference matches a stored procedure, type, or UDF in more than one schema in the path, the order of the schemas in the path is used to resolve this match. The SQL path is established by means of the FUNCPATH option on the precompile and bind commands for static SQL. The SQL path is set by the SET PATH statement for dynamic SQL. The SQL path has the following default value:

```
"SYSIBM","SYSFUN","SYSPROC", "ID"
```

This applies to both static and dynamic SQL, where *ID* represents the current statement authorization ID.

Routine names can be *overloaded*, which means that multiple routines, even in the same schema, can have the same name. Multiple functions or methods with the same name can have the same number of parameters, as long as the data types differ. This is not true for stored procedures, where multiple stored procedures with the same name must have different numbers of parameters. Instances of different routine types do not overload one-another, except for methods, which are able to overload functions. For a method to overload a function, the method must be registered using the WITH FUNCTION ACCESS clause.

A function, a stored procedure, and a method can have identical *signatures* and be in the same schema without overloading each other. In the context of routines, signatures are the qualified routine name concatenated with the defined data types of all the parameters in the order in which they are defined.

Methods are invoked against instances of their associated structured type. When a subtype is created, among the attributes it inherits are the methods defined for the supertype. Hence, a supertype's methods can also be run against any instances of its subtypes. When defining a subtype you can *override* the supertype's method. To override a method means to reimplement it specifically for a given subtype. This facilitates the dynamic dispatch of methods (also known as polymorphism), where an application will execute the most specific method depending on the type of the structured type instance (for example, where it is situated in the structured type hierarchy).

Each routine type has its own selection algorithm that takes into account the facts of overloading (in the case of methods, and overriding) and SQL path to choose the most appropriate match for every routine reference.

Related concepts:
- "User-Defined Structured Types" on page 200
- "Dynamic Dispatch of Methods" on page 207

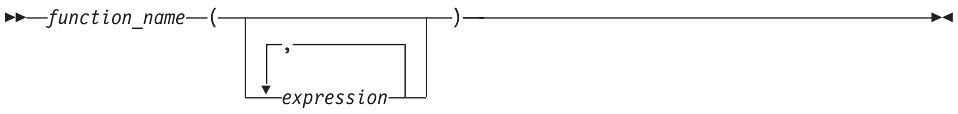
Related tasks:
- "Defining Behavior for Structured Types" on page 206

Related reference:
- "Functions" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "SET PATH statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

• "Methods" in the *SQL Reference, Volume 1*

References to Functions

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:



In the preceding syntax diagram, `function_name` can be either an unqualified or a qualified function name. The arguments can number from 0 to 90 and are expressions. Examples of some components that can compose expressions are the following:

- a column name, qualified or unqualified
- a constant
- a host variable
- a special register
- a parameter marker

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 that contain the columns have proper scope. For table functions referenced in a join and using any argument involving columns from another table or table function, the referenced table or table function must precede the table function containing the reference in the FROM clause.

In order to use parameter markers in functions you cannot simply code the following:

```
BLOOP(?)
```

Because 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))
```

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')
```

If any of the above functions are table functions, the syntax to reference them is slightly different than presented previously. For example, if PABLO.BLOOP is a table function, to properly reference it, use:

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

Related tasks:

- "Invoking UDFs" on page 148
- "Invoking User-Defined Table Functions" on page 149

Related reference:

- "Functions" in the *SQL Reference, Volume 1*

Function Selection

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 are functions in the named schema for a qualified reference, or functions in the schemas of the SQL path for an unqualified reference. The algorithm looks for an exact match, or failing that, a best match among these functions. The SQL 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.

You can nest function references, 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.

For 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.

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.

For a more thorough description of function selection, see the Function References section in the Functions topic listed in the related links.

Related concepts:

- "References to Functions" on page 156

Related tasks:

- "Invoking UDFs" on page 148
- "Invoking User-Defined Table Functions" on page 149

Related reference:

- "Functions" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Distinct Types as Routine Parameters

UDFs and methods can be defined with distinct types as parameters or as the result. DB2 will pass the value to the UDF or method in the format of the source data type of the distinct type.

Distinct type values that originate in a host variable and which are used as arguments to a UDF that 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. Consider the BOAT distinct type which is defined over a BLOB, and consider the BOAT_COST UDF defined as follows:

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  ...
```

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 SQL path. Your results may otherwise be ambiguous.

Related concepts:

- "User-Defined Types (UDTs) and Large Objects (LOBs)" in the *Application Development Guide: Programming Client Applications*
- "Function Selection" on page 157
- "Stored Procedure Selection" on page 162

Related tasks:

- "Passing Structured Type Parameters to External Routines" on page 257
- "LOB Values as UDF Parameters" on page 160

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "SELECT statement" in the *SQL Reference, Volume 2*

LOB Values as UDF Parameters

UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. DB2 will materialize the entire LOB value in storage before invoking such a function, even if the source of the value is a *LOB locator* host variable. For example, consider the following fragment of a C language application:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(150K) clob150K ;        /* LOB host var */
  SQL TYPE IS CLOB_LOCATOR clob_locator1;  /* LOB locator host var */
  char                   string[40];       /* string host var */
EXEC SQL END DECLARE SECTION;
```

Either host variable `:clob150K` or `:clob_locator1` is valid as an argument for a function whose corresponding parameter is defined as `CLOB(500K)`. For example, suppose you have registered a UDF as follows:

```
CREATE FUNCTION FINDSTRING (CLOB(500K, VARCHAR(200))
  ...
```

Both of the following invocations of FINDSTRING 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 SQL to manipulate the actual bytes of the LOB value.

You can also use this capability on UDF parameters or results which have a distinct type that is based on a LOB. 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.

Related concepts:

- "User-Defined Types (UDTs) and Large Objects (LOBs)" in the *Application Development Guide: Programming Client Applications*
- "Function Selection" on page 157
- "Stored Procedure Selection" on page 162

Related tasks:

- "Retrieving a LOB Value with a LOB Locator" on page 168
- "Distinct Types as Routine Parameters" on page 159

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

References to Stored Procedures

Stored Procedures are invoked from the CALL statement where they are referenced by a qualified name (schema and stored procedure name), followed by a list of arguments enclosed by parentheses. A stored procedure can also be invoked without the schema name, resulting in a choice of possible stored procedures in different schemas with the same number of parameters.

Each parameter passed to the stored procedure can be composed of a host variable, parameter marker, expression, or NULL. The following are restrictions for stored procedure parameters:

- OUT and INOUT parameters must be host variables.
- NULLs cannot be passed to Java™ stored procedures unless the SQL data type maps to a Java class type.
- NULLs cannot be passed to PARAMETER STYLE GENERAL stored procedures.

The position of the arguments is important and must conform to the stored procedure definition for the semantics to be correct. Both the position of the arguments and the stored procedure definition must conform to the stored procedure body itself. DB2® does not attempt to shuffle arguments to better match a stored procedure definition, and DB2 does not understand the semantics of the individual stored procedure parameters.

Related concepts:

- “Parameter Styles for External Routines” on page 71

Related tasks:

- “Invoking Stored Procedures” on page 146

Related reference:

- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*
- “Syntax for Passing Arguments to Routines Written in C/C++, OLE, or COBOL” on page 74

Stored Procedure Selection

Given a stored procedure invocation, the database manager must decide which of the possible stored procedures with the same name to call. Stored procedure resolution is done using the steps that follow.

1. Find all stored procedures from the catalog (SYSCAT.ROUTINES), such that all of the following are true:
   - For invocations where the schema name was specified (that is, qualified references), the schema name and the stored procedure name match the invocation name.
   - For invocations where the schema name was not specified (that is, unqualified references), the stored procedure name matches the invocation name, and has a schema name that matches one of the schemas in the SQL path.
   - The number of defined parameters matches the invocation.
   - The invoker has the EXECUTE privilege on the stored procedure.
2. Choose the stored procedure whose schema is earliest in the SQL path.

If there are no candidate stored procedures remaining after the first step, an error is returned (SQLSTATE 42884).

Related concepts:

- “References to Stored Procedures” on page 161

Related tasks:

- “Invoking Stored Procedures” on page 146

Part 2. Large Objects, User-Defined Distinct Types, and Triggers

Chapter 6. Large Objects

Large Object Usage

The VARCHAR and 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 gigabytes (GB) in size. The three large object (LOB) data types are Binary Large Objects (BLOBs), Character Large Objects (CLOBs), and Double-Byte Character Large Objects (DBCLOBs).

Note: CLOBs can contain either single-byte or double-byte characters. DBCLOBs can contain either four-byte or double byte characters.

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 2 terabytes of LOB data.

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.

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.

With their potentially 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 approaching 1GB 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 will improve performance, changes to the LOB values after the most recent backup are lost during roll-forward recovery.

When selecting a LOB value, you have three options:

1. Select the entire LOB value into a host variable. The entire LOB value is copied from the server to the client. This is inefficient and is sometimes not feasible. Host variables use the client memory buffer, which may not have the capacity to hold larger LOB values.
2. Select just a LOB locator into a host variable. The LOB value remains on the server; the LOB locator moves to the client. 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.
3. Select the entire LOB value into a file reference variable. The LOB value (or a part of it) is moved to a file at the client without going through the application's memory.

Related concepts:

- "Large Object Locators" on page 166
- "Large Object File Reference Variables" on page 173

Large Object Locators

A large object locator (or LOB locator) is a host variable with a 4-byte value that represents a single LOB value in the database server. LOB locators provide users with a mechanism by which they can easily manipulate very large objects in application programs without requiring them to store the entire LOB value on the client machine where the application program may be running. Subsequent statements can then use the locators to perform operations on the data without necessarily retrieving the entire large object. Locator variables are used to reduce the storage requirements for applications that access LOBs, and improve their performance by reducing the flow of data between the client and the server.

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.

LOB locators can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT
        <lob 3>, <start>, <length> )
```

It is important to understand that a LOB locator represents a value, not a row or location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value that is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data in order to provide this function. Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown above) is deferred until it is actually assigned to some location -- either into a user buffer in the form of a host variable or into another record's field value in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. 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. Furthermore, a LOB locator is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, since a LOB locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN and EXECUTE statements. They can also be passed between DB2® and UDFs.

For normal host variables in an application program, when selecting a NULL value into a host variable, the indicator variable is set to -1, 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.

Related concepts:
- “Large Object Usage” on page 165

Related tasks:
- “Retrieving a LOB Value with a LOB Locator” on page 168
- “Deferring the Evaluation of LOB Expressions” on page 170

Related samples:
- “dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)”
- “dtlob.sqc -- How to use the LOB data type (C)”
- “dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)”
- “dtlob.sqC -- How to use the LOB data type (C++)”
- “dtLob.bas -- Get/set Large Objects (LOBs)”
- “DtLob.java -- How to use LOB data type (JDBC)”
- “DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)”
- “lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)”
- “lobloc.sqb -- Demonstrates the use of LOB locators (IBM COBOL)”

Retrieving a LOB Value with a LOB Locator

If you need to extract data from a LOB you can use LOB locators. This is a good alternative if the LOB to be accessed is large. Transferring the entire LOB to a client when only a subset of the LOB data is needed is avoided with the use of locators.

The example uses embedded SQL in C.

Procedure:

To retrieve a LOB value with a LOB locator:

1. Declare the LOB locator host variables:
   ```
   EXEC SQL BEGIN DECLARE SECTION;
      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;
```

In the host variable declaration section:

- `number` will contain the value returned by `empno` in the SELECT statement to be issued by the cursor `c1`.
- `deptInfoBeginLoc` and `deptInfoEnd` will temporarily hold LOB locator values.
- `resume` and `deptBuffer` are LOB locators.
- `lobind` is used to indicate if the LOB read is null or not.
- `buffer` will contain the data extracted from the LOB.
- `userid` and `passwd` represent a userid and password combination, which are needed for the application to connect to a database.

2. Connect the application to the database.
3. Declare and open a cursor:

```
EXEC SQL DECLARE c1 CURSOR FOR
   SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
   AND empno <> 'A00130';

EXEC SQL OPEN c1;
```

4. Fetch the LOB value into the host variable locator.

```
EXEC SQL FETCH c1 INTO :number, :resume :lobind;
```

5. Evaluate the LOB locator:
   a. Locate the beginning of `Department Information` section:

   ```
   EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
      INTO :deptInfoBeginLoc;
   ```

   b. Locate the beginning of `Education` section (end of `Department Information`):

   ```
   EXEC SQL VALUES (POSSTR(:resume, 'Education'))
      INTO :deptInfoEndLoc;
   ```

   c. Obtain only the `Department Information` section by using SUBSTR:

   ```
   EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
      :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
   ```

   d. Append the `Department Information` section to the `:buffer` variable:

   ```
   EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
   ```

6. Free the LOB locators `resume` and `deptBuffer`:

```
EXEC SQL FREE LOCATOR :resume, :deptBuffer;
```

7. Close the cursor:

```
EXEC SQL CLOSE c1;
```

8. End the program.

Related concepts:

- "Large Object Usage" on page 165
- "Large Object Locators" on page 166

Related tasks:

- "Connecting an Application to a Database" in the *Application Development Guide: Programming Client Applications*
- "Ending an Application Program" in the *Application Development Guide: Programming Client Applications*
- "Deferring the Evaluation of LOB Expressions" on page 170

Related samples:

- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "dtLob.bas -- Get/set Large Objects (LOBs)"
- "DtLob.java -- How to use LOB data type (JDBC)"
- "DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)"
- "lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)"
- "lobloc.sqb -- Demonstrates the use of LOB locators (IBM COBOL)"

Deferring the Evaluation of LOB Expressions

The bytes of a LOB value do not move until you assign 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 technique is known as deferring the evaluation of a LOB expression.

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 and usually require fewer disk I/Os.

The example uses embedded SQL in C.

Procedure:

To defer the evaluation of a LOB expression:

1. Declare the LOB locator host variables:

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

In the host variable declaration section:
- `hv_start_deptinfo`, `hv_return_code`, and `hv_start_educ` will temporarily hold LOB locator values.
- `hv_new_section_buffer` will contain the data extracted from the LOB.
- `hv_doc_locator1`, `hv_doc_locator2`, and `hv_doc_locator3` are LOB locators.
- `userid` and `passwd` represent a userid and password combination, which are needed for the application to connect to a database.

2. Connect the application to the database.
3. Fetch the LOB value into the host variable locator:

```
EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
   WHERE empno = '000130' AND resume_format = 'ascii';
```

4. Manipulate LOB data with locators. These five statements manipulate LOB data without moving the actual data contained in the LOB field.
   a. Use the POSSTR function to locate the start of the `Department Information` section:

   ```
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
      INTO :hv_start_deptinfo;
   ```

   b. Use the POSSTR function to locate the start of the `Education` section:

   ```
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
      INTO :hv_start_educ;
   ```

   c. Replace the `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;
   ```

   d. Move the `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;
   ```

   e. Append the new section to the end. Effectively, this just moves the `Department Information` section to the bottom of the resume.

```
    EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer)
       INTO :hv_doc_locator3;
```

5. Move LOB data to the target destination:

   ```
   EXEC SQL INSERT INTO emp_resume
                   VALUES ('A00130', 'ascii', :hv_doc_locator3);
   ```

   The evaluation of the LOB assigned to the target destination is postponed until this statement. It is at this point that LOB value bytes finally move.
6. Free the LOB locators `hv_doc_locator1`, `hv_doc_locator2`, and `hv_doc_locator3`:

   ```
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2,
                         : hv_doc_locator3;
   ```
7. End the program.

In this example, a particular resume (empno = '000130') was sought from within a table of resumes EMP_RESUME. The `Department Information` section of the resume was copied, cut, and then appended to the end of the resume. This new resume was then inserted into the EMP_RESUME table. The original resume in this table was left unchanged.

Locators permitted 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.

Related concepts:
- "Large Object Usage" on page 165
- "Large Object Locators" on page 166

Related tasks:
- "Connecting an Application to a Database" in the *Application Development Guide: Programming Client Applications*
- "Ending an Application Program" in the *Application Development Guide: Programming Client Applications*
- "Retrieving a LOB Value with a LOB Locator" on page 168

Related samples:
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "dtLob.bas -- Get/set Large Objects (LOBs)"
- "DtLob.java -- How to use LOB data type (JDBC)"

- “DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)”
- “lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)”

Large Object File Reference Variables

LOB file reference variables facilitate the movement of LOB values from the database server to a client application without going through the client application’s memory. File reference variables are similar to host variables except that they are used to transfer data to and from client files, and not to and from memory buffers. With this approach, client applications do not have to call utility routines to read and write files using host variables (which have size restrictions) to carry out the movement of LOB data.

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 column values.

File reference variables are used for direct file input and output for LOBs, and can be defined in all host languages. Since they are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

A file reference variable has the following properties:

1. Data Type: BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared.
2. File name: The application program must specify this at run time. It is one of:
   - The complete path name of the file (which is advised).
   - A relative file name. If a relative file name is provided, it is appended to the current path of the client process. Within an application, a file should only be referenced in one file reference variable.
3. File Name Length: The application program must specify this at run time. It is the length of the file name (in bytes).
4. File Options: An application must assign one of a number of options to a file reference variable before it makes use of that variable. Options are set by an INTEGER value in a field in the file reference variable structure. One of the file options must be specified for each file reference variable:

File option (by language)	Direction	Description
C: SQL_FILE_READ COBOL: SQL-FILE-READ FORTRAN: sql_file_read	input	This is a regular file that can be opened, read and closed.

File option (by language)	Direction	Description
C: SQL_FILE_CREATE COBOL: SQL-FILE-CREATE FORTRAN: sql_file_create	output	Create a new file. If the file already exists, an error is returned.
C: SQL_FILE_OVERWRITE COBOL: SQL-FILE-OVERWRITE FORTRAN: sql_file_overwrite	output	If an existing file with the specified name exists, it is overwritten; otherwise, a new file is created.
C: SQL_FILE_APPEND COBOL: SQL-FILE-APPEND FORTRAN: sql_file_append	output	If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created.

5. Data Length: This is unused on input. On output, the implementation sets the data length (in bytes) to the length of the new data written to the file.

For normal host variables in an application program, when selecting a NULL value into a host variable, the indicator variable is set to -1, signifying that the value is NULL. In the case of file reference variables, however, the meaning of indicator variables is slightly different. Since a file reference variable itself can never be NULL, a negative indicator variable value indicates that the LOB value represented by the file reference variable is NULL.

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.

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.

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.

Related concepts:

- “Large Object Usage” on page 165

Related tasks:

- “Writing Data from a CLOB Column to a Text File” on page 175
- “Inserting Data from a Text File into a CLOB Column” on page 176

Related samples:

- “dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)”
- “dtlob.sqc -- How to use the LOB data type (C)”
- “dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)”
- “dtlob.sqC -- How to use the LOB data type (C++)”
- “dtLob.bas -- Get/set Large Objects (LOBs)”
- “DtLob.java -- How to use LOB data type (JDBC)”
- “DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)”
- “lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)”

Writing Data from a CLOB Column to a Text File

If you need access to data in a CLOB column outside of the database, write it to a text file.

The example in the procedure uses embedded SQL in C. In this example, a particular resume (empno = ’000130’) is SELECTed from a CLOB column and put into a text file.

Procedure:

To write data from a CLOB column to a text file:

1. Declare the CLOB FILE REFERENCE host variable:

   ```
   EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE IS CLOB_FILE resume;
      char userid[9];
      char passwd[19];
      short lobind;
   EXEC SQL END DECLARE SECTION;
   ```

   In the host variable declaration section:
   - `resume` represents the file that will contain the data extracted from the CLOB column.
   - `userid` and `passwd` represent a userid and password combination, which are needed for the application to connect to a database.
2. Connect the application to the database.
3. Set up the CLOB FILE REFERENCE host variable:

   ```
   strcpy (resume.name, "RESUME.TXT");
   resume.name_length = strlen("RESUME.TXT");
   resume.file_options = SQL_FILE_OVERWRITE;
   ```

   In the path description provided in the strcpy function:
   - `RESUME.TXT` is the name of the file whose data will be inserted into the table.

4. SELECT the data from the resume field in the CLOB column into the specified text file.

   ```
   EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume
      WHERE resume_format='ascii' AND empno='000130';
   ```
5. End the program.

Related concepts:
- "Large Object Locators" on page 166
- "Large Object File Reference Variables" on page 173

Related tasks:
- "Connecting an Application to a Database" in the *Application Development Guide: Programming Client Applications*
- "Ending an Application Program" in the *Application Development Guide: Programming Client Applications*
- "Inserting Data from a Text File into a CLOB Column" on page 176

Related samples:
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "dtLob.bas -- Get/set Large Objects (LOBs)"
- "DtLob.java -- How to use LOB data type (JDBC)"
- "DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)"
- "lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)"

Inserting Data from a Text File into a CLOB Column

If you need the database to process CLOB data that currently exists in a text file, insert it into a CLOB column.

The example uses embedded SQL in C on a UNIX-based file system.

Procedure:

To insert data from a text file into a CLOB column:
1. Declare the CLOB FILE REFERENCE host variable:

   ```
     EXEC SQL BEGIN DECLARE SECTION;
       SQL TYPE IS CLOB_FILE hv_text_file;
     EXEC SQL END DECLARE SECTION;
   ```

`hv_text_file` represents a file.

2. Connect the application to the database.
3. Set up the CLOB FILE REFERENCE host variable:

```
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;
```

   In the path description provided in the strcpy function:
   - `userid` represents the directory for one of your users.
   - `dirname` represents a subdirectory belonging to "userid".
   - `filnam.1` is the name of the file whose data will be inserted into the table.
   - `clobtab` is the name of the table with the CLOB data type.
4. Insert data from `hv_text_file` into the CLOB table.

```
EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

5. End the program.

Related concepts:

- "Large Object Locators" on page 166
- "Large Object File Reference Variables" on page 173

Related tasks:

- "Connecting an Application to a Database" in the *Application Development Guide: Programming Client Applications*
- "Ending an Application Program" in the *Application Development Guide: Programming Client Applications*
- "Writing Data from a CLOB Column to a Text File" on page 175

Related samples:

- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "dtLob.bas -- Get/set Large Objects (LOBs)"
- "DtLob.java -- How to use LOB data type (JDBC)"
- "DtLob.out -- HOW TO USE LOB DATA TYPE (JDBC)"
- "lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)"

Chapter 7. User-Defined Distinct Types

User-Defined Types

A user-defined type (UDT) is a data type that you derive from existing data types, but is nevertheless considered to be separate and incompatible from them. UDTs enable you to extend the built-in types already available in DB2® and create your own customized data types.

There are two classifications of user-defined types:

- distinct type: shares a common representation with built-in data types.
- structured type: enables the representation of a sequence of named attributes that each have a type. One structured type can be a subtype of another structured type (called a supertype), defining a type hierarchy.

Related concepts:

- “User-Defined Distinct Types” on page 179
- “User-Defined Structured Types” on page 200

Related tasks:

- “Defining Distinct Types” on page 182

User-Defined Distinct Types

Distinct types are user-defined types that are based on existing DB2® built-in data types. Internally, a distinct type shares its representation with an existing type (the source type), but is considered to be a separate and incompatible type.

For example, distinct types can represent various currencies, such as US_Dollar 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.

DB2 also has built-in types for storing and manipulating 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. The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type that is not comparable to a BLOB or to any other type. This allows the creation of functions written specifically for AUDIO and assures that these functions will not be applied to any other type.

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.
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.

Distinct types are identified by qualified identifiers. If the schema name is not used to qualify the distinct type name when used in statements other than CREATE DISTINCT TYPE, DROP DISTINCT TYPE, or COMMENT ON DISTINCT TYPE, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types sourced on LONG VARCHAR, LONG VARGRAPHIC, LOB types, or DATALINK are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type by defining user-defined functions. (These functions are sourced on functions defined on the source type of the distinct type.) The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, or DATALINK as the source type. In addition, functions are generated to support casting from the source type to the distinct type, and from the distinct type to the source type.

Related concepts:

- "Strong Typing in User-Defined Distinct Types" on page 181
- "User-Defined Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184
- "Manipulating Distinct Types" on page 189

Related samples:

- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)"
- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP (SQLJ)"
- "DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)"

Strong Typing in User-Defined Distinct Types

One of the most important concepts associated with distinct types is strong typing. Strong typing guarantees that only functions and operators defined explicitly 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.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184

Defining Distinct Types

A user-defined distinct type is a data type derived from an existing type, such as an integer, decimal, or character type. When you create distinct types, DB2 generates cast functions to cast from the distinct type to the source type, and to cast from the source type to the distinct type. These functions are essential for the manipulation of distinct types in queries.

Instances of the same distinct type can be compared to each other, if the WITH COMPARISONS clause is specified on the CREATE DISTINCT TYPE statement (as in the example in the procedure). The WITH COMPARISONS clause cannot be specified if the source data type is a large object, a DATALINK, LONG VARCHAR, or LONG VARGRAPHIC type.

Prerequisites:

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

Restrictions:

The source type of the distinct type is the data 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.

Procedure:

To define a distinct type, issue the CREATE DISTINCT TYPE statement, specifying a type name and the source type. For example, the following statement defines a new distinct type called new_type, that contains SMALLINT values:

```
   CREATE DISTINCT TYPE new_type AS SMALLINT WITH COMPARISONS
```

Because the distinct type defined in the above statement is based on SMALLINT, the WITH COMPARISONS parameters must be specified.

To further understand the application of user-defined distinct types, see the following examples of distinct type definitions based on sample business cases:

- Define currency-based distinct types.
- Define a distinct type for job applications.

Related concepts:

- "Strong Typing in User-Defined Distinct Types" on page 181
- "User-Defined Types" on page 179
- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Currency-Based Distinct Types" on page 186
- "Defining a Distinct Type for Completed Job Application Forms" on page 187
- "Manipulating Distinct Types" on page 189

Related reference:

- "CREATE DISTINCT TYPE statement" in the *SQL Reference, Volume 2*

Related samples:

- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)"
- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP (SQLJ)"
- "DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)"

Creating Tables with Columns Based on Distinct Types

After you have defined distinct types, you can start creating tables with columns based on distinct types.

Prerequisites:

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

For the list of privileges required to create tables, see the CREATE TABLE statement.

Procedure:

To create a table with columns based on distinct types:

1. Define a distinct type:
   ```
   CREATE DISTINCT TYPE t_educ AS SMALLINT WITH COMPARISONS
   ```
2. Create the table, naming the distinct type, T_EDUC as a column type.
   ```
   CREATE TABLE employee
     (empno CHAR(6) NOT NULL,
      firstnme VARCHAR(12) NOT NULL,
      lastname VARCHAR(15) NOT NULL,
      workdept CHAR(3),
      phoneno CHAR(4),
      photo BLOB(10M) NOT NULL,
      edlevel T_EDUC)
     IN RESOURCE
   ```

To further understand the application of tables, see the following examples of table creation based on sample business cases:

- Create tables to track international sales.
- Create a table to store filled job application forms.

Related concepts:

- “Strong Typing in User-Defined Distinct Types” on page 181
- “User-Defined Types” on page 179
- “User-Defined Distinct Types” on page 179

Related tasks:

- “Creating Tables to Track International Sales” on page 188
- “Creating a Table to Store Completed Job Application Forms” on page 189
- “Defining Distinct Types” on page 182
- “Manipulating Distinct Types” on page 189

- “Defining Currency-Based Distinct Types” on page 186
- “Defining a Distinct Type for Completed Job Application Forms” on page 187

Related reference:

- “CREATE DISTINCT TYPE statement” in the *SQL Reference, Volume 2*
- “CREATE TABLE statement” in the *SQL Reference, Volume 2*

Dropping User-Defined Types

You can drop a user-defined type (UDT) using the DROP statement. 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.
- As the supertype of another structured type.

The database manager attempts to drop every routine that is dependent on this UDT. A routine cannot be dropped if a view, trigger, table check constraint, or another routine is dependent on it. If DB2 cannot drop a dependent routine, 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. Note that you can only drop user-defined transforms. You cannot drop built-in transforms or their associated group definitions.

Related concepts:

- “User-Defined Types” on page 179
- “User-Defined Distinct Types” on page 179
- “User-Defined Structured Types” on page 200
- “Transform Functions and Transform Groups” on page 246

Related tasks:

- “Defining Distinct Types” on page 182
- “Defining Structured Types” on page 201

Related reference:

- “DROP statement” in the *SQL Reference, Volume 2*

Related samples:

- “dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)”
- “dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)”
- “dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)”
- “dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)”
- “dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)”
- “dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)”
- “DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)”
- “DtUdt.out -- HOW TO CREATE, USE AND DROP (JDBC)”
- “DtUdt.out -- HOW TO CREATE, USE AND DROP (SQLJ)”
- “DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)”

Defining Currency-Based Distinct Types

Suppose you are writing applications that need to handle different currencies. Given that conversions are necessary whenever you want to compare values of different currencies, you want to ensure that DB2 does not allow these currencies to be compared or manipulated directly with one another. Because distinct types are only compatible with themselves, you must define one for each currency that you need to represent.

Prerequisites:

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

Procedure:

To define distinct types representing the euro and the American and Canadian currencies, issue the following statements:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,3) WITH COMPARISONS
```

Note that you have to specify the WITH COMPARISONS clause because comparison operators are supported on DECIMAL (9,3).

Related concepts:

- “User-Defined Distinct Types” on page 179

Related tasks:

- “Creating Tables with Columns Based on Distinct Types” on page 184
- “Defining a Distinct Type for Completed Job Application Forms” on page 187
- “Creating Tables to Track International Sales” on page 188

Related reference:
- “CREATE DISTINCT TYPE statement” in the *SQL Reference, Volume 2*

Defining a Distinct Type for Completed Job Application Forms

Suppose you would like to keep incoming job application forms in a DB2 table and be able to use functions to extract the information from these forms. You can define a distinct type to represent the forms in tables and as parameters to functions.

Prerequisites:

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

Procedure:

To define a distinct type representing the completed job application forms, issue the following statement:

```
CREATE DISTINCT TYPE PERSONNEL.APPLICATION_FORM AS CLOB(32K)
```

Because DB2 does not support comparisons on CLOBs, you cannot specify the WITH COMPARISONS clause. The PERSONNEL schema is specified in the above statement because the schema intended to contain all the distinct types and UDFs dealing with application forms.

Related concepts:
- “User-Defined Distinct Types” on page 179

Related tasks:
- “Creating Tables with Columns Based on Distinct Types” on page 184
- “Defining Currency-Based Distinct Types” on page 186
- “Creating a Table to Store Completed Job Application Forms” on page 189

Creating Tables to Track International Sales

Suppose you want to define tables to track your company’s sales in different regions. You can create tables using the applicable currency distinct type as the column type for a given region’s total sales revenue.

Prerequisites:

For the list of privileges required to create tables, see the CREATE TABLE statement.

Procedure:

To create tables to track international sales:

1. Create currency-based distinct types.
2. Issue the following CREATE TABLE statements:

```
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)
```

Related concepts:

- “User-Defined Distinct Types” on page 179

Related tasks:

- “Defining Distinct Types” on page 182
- “Defining Currency-Based Distinct Types” on page 186
- “Creating a Table to Store Completed Job Application Forms” on page 189

Creating a Table to Store Completed Job Application Forms

Suppose you need to define a table where you keep the forms filled out by applicants. You can create a table using the distinct type PERSONNEL.APPLICATION_FORM as a column type to contain the completed forms.

Prerequisites:

For the list of privileges required to create tables, see the CREATE TABLE statement.

Procedure:

To create a table to contain completed job application forms:

1. Create a distinct type for a job application form.
2. Issue the following CREATE TABLE statement:

```
CREATE TABLE APPLICATIONS
  (ID                SYSIBM.INTEGER,
   NAME              VARCHAR (30),
   APPLICATION_DATE  SYSIBM.DATE,
   FORM              PERSONNEL.APPLICATION_FORM)
```

The distinct type name is fully qualified because its qualifier is not the same as the authorization ID and the default function path was not changed. 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.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Defining a Distinct Type for Completed Job Application Forms" on page 187
- "Creating Tables to Track International Sales" on page 188

Manipulating Distinct Types

Manipulating Distinct Types

Once you define distinct types and create tables based upon them, you can begin manipulating actual distinctly typed values.

Procedure:

To implement various kinds of distinct type manipulation:

- Cast between distinct types.
- Perform comparisons between distinct types.
- Perform comparisons between distinct types and constants.
- Define sourced UDFs for distinct types.
- Perform assignments involving distinct types.
- Perform assignments involving distinct types in dynamic SQL.
- Perform assignments involving different distinct types.
- Perform UNION operations on distinctly typed columns.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Casting between Distinct Types" on page 191
- "Performing Comparisons Involving Distinct Types" on page 192
- "Performing Comparisons between Distinct Types and Constants" on page 193
- "Defining Sourced UDFs for Distinct Types" on page 196
- "Performing Assignments Involving Distinct Types in Embedded SQL" on page 193
- "Performing Assignments Involving Distinct Types in Dynamic SQL" on page 194
- "Performing Assignments Involving Different Distinct Types" on page 195
- "Performing UNION Operations on Distinctly Typed Columns" on page 196
- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184

Related samples:

- "dtudt.c -- How to create, use, and drop user-defined distinct types. (CLI)"
- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)"
- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)"

- “DtUdt.out -- HOW TO CREATE, USE AND DROP (JDBC)”
- “DtUdt.out -- HOW TO CREATE, USE AND DROP (SQLJ)”
- “DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)”

Casting between Distinct Types

Suppose you want to define a UDF that converts another currency into U.S. dollars. For the purposes of this example, you can obtain the current exchange rate from a table such as the following:

```
CREATE TABLE
  exchange_rates(source CHAR(3), target CHAR(3), rate DECIMAL(9,3))
```

The following function can be used to directly access the values in the exchange_rates table:

```
CREATE FUNCTION exchange_rate(src VARCHAR(3), trg VARCHAR(3))
  RETURNS DECIMAL(9,3)
  RETURN SELECT rate FROM exchange_rates
    WHERE source = src AND target = trg
```

The currency exchange rates in the above function are based on the DECIMAL type, not distinct types. To represent some different currencies, use the following distinct type definitions:

```
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL(9,3) WITH COMPARISONS
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
```

To create a UDF that converts CANADIAN_DOLLAR or EURO to US_DOLLAR you need to cast the values involved. Note that the exchange_rate function returns an exchange rate as a DECIMAL. For example, a function that converts values of CANADIAN_DOLLAR to US_DOLLAR performs the following steps:

- cast the CANADIAN_DOLLAR value to DECIMAL
- get the exchange rate for converting the Canadian dollar to the U.S. dollar from the exchange_rate function, which returns the exchange rate as a DECIMAL value
- multiply the Canadian dollar DECIMAL value to the DECIMAL exchange rate
- cast this DECIMAL value to US_DOLLAR
- return the US_DOLLAR value

The following are instances of the US_DOLLAR function (for both the Canadian dollar and the euro), which follow the above steps.

```
CREATE FUNCTION US_DOLLAR(amount CANADIAN_DOLLAR)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('CAN', 'USD'))
```

```
CREATE FUNCTION US_DOLLAR(amount EURO)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('EUR', 'USD'))
```

Related concepts:
- “User-Defined Distinct Types” on page 179

Related tasks:
- “Defining Distinct Types” on page 182
- “Creating Tables with Columns Based on Distinct Types” on page 184
- “Defining Sourced UDFs for Distinct Types” on page 196

Performing Comparisons Involving Distinct Types

Suppose you want to know which products sold more in the United States 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 U.S. dollars with Canadian dollars or euros, use the UDF to cast the amount in Canadian dollars to US dollars, and the UDF to cast the amount in euros to U.S. dollars. You should not 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.

Related concepts:
- “User-Defined Distinct Types” on page 179

Related tasks:
- “Defining Distinct Types” on page 182
- “Creating Tables with Columns Based on Distinct Types” on page 184
- “Casting between Distinct Types” on page 191

Performing Comparisons between Distinct Types and Constants

Suppose you want to know which products sold more than U.S. $100 000.00 in the United States 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 U.S. dollars (that is, DECIMAL) directly, you have used the cast function provided by DB2 to cast from DECIMAL to U.S. dollars. You can also use the other cast function provided by DB2 (that is, the one to cast from U.S. 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
```

Related concepts:

- “User-Defined Distinct Types” on page 179

Related tasks:

- “Defining Distinct Types” on page 182
- “Creating Tables with Columns Based on Distinct Types” on page 184
- “Casting between Distinct Types” on page 191

Performing Assignments Involving Distinct Types in Embedded SQL

Suppose you want to store the job application form completed by a new applicant into the database. You can define a host variable containing the character string value used to represent the completed 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 host variable 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.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184
- "Defining Sourced UDFs for Distinct Types" on page 196

Performing Assignments Involving Distinct Types in Dynamic SQL

Suppose you want to store the job application form completed by a new applicant into the database. You have defined a host variable containing the character string value used to represent the completed form. To use 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;
```

This makes 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, since host languages do not support distinct types. Therefore, you cannot specify that the type of a parameter marker is a distinct type.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184

Performing Assignments Involving Different Distinct Types

Suppose you have defined two sourced UDFs on the built-in SUM function to support SUM on U.S. and Canadian dollars:

```
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 U.S. dollars of each product and in each region, 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 convert 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.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:
- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184

Performing UNION Operations on Distinctly Typed Columns

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. The above example makes use of the UDFs defined in Casting between Distinct Types to cast between the currencies, which results in the use of functional notation instead of a cast specification.

Related concepts:
- "User-Defined Distinct Types" on page 179

Related tasks:
- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184
- "Casting between Distinct Types" on page 191

Defining Sourced UDFs for 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.

Related concepts:

- "User-Defined Distinct Types" on page 179

Related tasks:

- "Defining Distinct Types" on page 182
- "Creating Tables with Columns Based on Distinct Types" on page 184
- "Performing Assignments Involving Distinct Types in Embedded SQL" on page 193

Chapter 8. User-Defined Structured Types

User-Defined Structured Types

A structured type is a user-defined data type containing one or more named attributes, each of which has a data type. Attributes are properties that describe an instance of a type. A geometric shape, for example, might have attributes such as its list of Cartesian coordinates. A person might have attributes of name, address, and so on. A department might have attributes of a name or some other kind of ID.

A structured type also includes a set of method specifications. Methods enable you to define behaviors for structured types. Like user-defined functions (UDFs), methods are routines that extend SQL. In the case of methods, however, the behavior is integrated solely with a particular structured type.

A structured type may be used as the type of a table, view, or column. When used as the type for a table or view, that table or view is known as a typed table or typed view respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of the typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type.

A type cannot be dropped when certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes a direct or indirect use of the type.

Related concepts:

- “User-Defined Types” on page 179
- “Typed Tables” on page 211
- “Typed Views” on page 228

Related tasks:

- “Defining Structured Types” on page 201
- “Storing Instances of Structured Types” on page 202
- “Defining Behavior for Structured Types” on page 206
- “Dropping User-Defined Types” on page 185

Related samples:

- “dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)”
- “dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)”

Defining Structured Types

A structured type is a user-defined type that contains one or more attributes, each of which has a name and a data type of its own. A structured type can serve as the type of a table or view in which each column of the table derives its name and data type from one of the attributes of the structured type. A structured type can also serve as a type of a column or a type for an argument to a routine.

Prerequisites:

For the list of privileges required to define structured types, see the CREATE TYPE statement.

Procedure:

To define a structured type to represent a person, with age and address attributes, issue the following statement:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   INSTANTIABLE
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;
```

Unlike distinct types, the attributes of structured types can be composed of types other than the built-in DB2 data types. The above type declaration includes an attribute called `Address` whose source type is another structured type, `Address_t`.

Related concepts:

- “User-Defined Distinct Types” on page 179
- “User-Defined Structured Types” on page 200
- “Structured Type Hierarchies” on page 203

Related tasks:

- “Storing Instances of Structured Types” on page 202
- “Creating a Structured Type Hierarchy” on page 204
- “Dropping User-Defined Types” on page 185

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

Related samples:

- "dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)"
- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"

Storing Instances of Structured Types

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. If you need to refer to an instance from other tables, you must use typed tables. 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)
     ...
  ```

Related concepts:

- "User-Defined Structured Types" on page 200
- "Typed Tables" on page 211

Related tasks:

- "Storing Objects in Typed Table Rows" on page 218
- "Storing Structured Type Objects in Table Columns" on page 237

Instantiability in Structured Types

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.

Related concepts:

- "User-Defined Structured Types" on page 200

Related tasks:

- "Defining Structured Types" on page 201

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

Structured Type Hierarchies

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. The original type is the *supertype*. 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`.

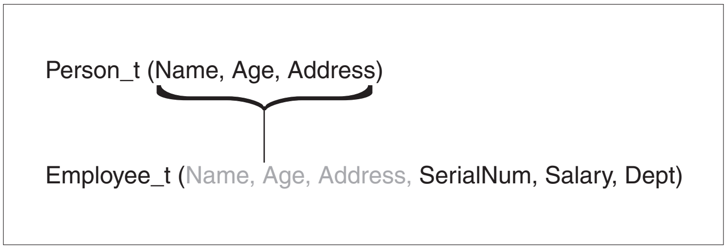

Person_t (Name, Age, Address)
Employee_t (Name, Age, Address, SerialNum, Salary, Dept)

Figure 1. Structured type Employee_t inherits attributes from supertype Person_t

A set of subtypes based (at some level) on the same supertype is known as a type hierarchy. 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 following figure presents an illustration of the various subtypes that might be derived from person and employee types:

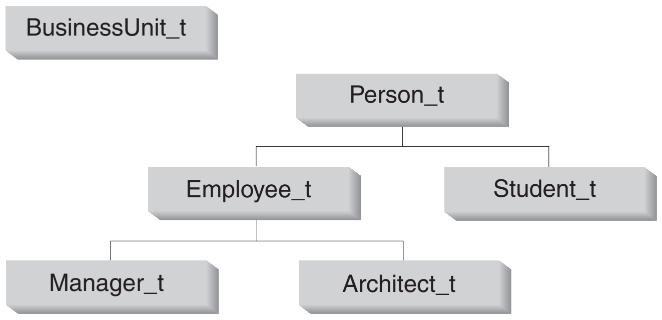


Figure 2. Type hierarchies (BusinessUnit_t and Person_t)

In Figure 2, 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`.

The department type, `BusinessUnit_t` is considered a trivial type hierarchy. It is the root of a hierarchy with no subtypes.

Related concepts:

- "User-Defined Structured Types" on page 200

Related tasks:

- "Defining Structured Types" on page 201
- "Creating a Structured Type Hierarchy" on page 204

Creating a Structured Type Hierarchy

The following figure presents an illustration of a structured type hierarchy:

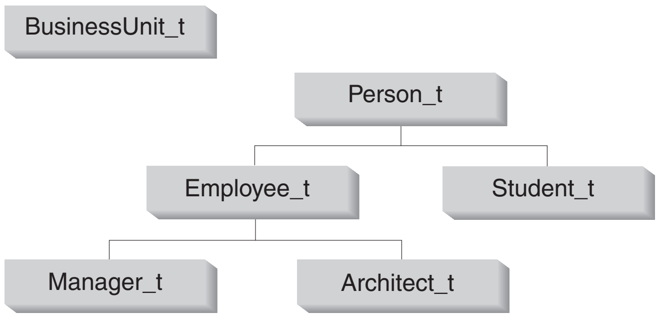


Figure 3. Type hierarchies (BusinessUnit_t and Person_t)

To create the `BusinessUnit_t` type, issue the following CREATE TYPE SQL statement:

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

To create the `Person_t` type hierarchy, issue the following SQL statements:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   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.

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`.

Related concepts:
- "User-Defined Structured Types" on page 200
- "Structured Type Hierarchies" on page 203

Related tasks:
- "Defining Structured Types" on page 201
- "Creating Typed Tables" on page 212

Related reference:
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

Related samples:
- "dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)"
- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"

Defining Behavior for Structured 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)
   RETURNS DECIMAL(7,2)
   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.

Related concepts:

- "User-Defined Structured Types" on page 200
- "Dynamic Dispatch of Methods" on page 207

Related tasks:

- "Defining Structured Types" on page 201

Related reference:

- "ALTER TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

Dynamic Dispatch of Methods

The behavior for a structured type is represented by its methods. These methods can only be invoked against instances of their structured type. When a subtype is created, among the attributes it inherits are the methods defined for the supertype. Hence, a supertype's methods can also be run against any instances of its subtypes.

If you do not want a method defined for a supertype to be used for a particular subtype, you can override the method. To override a method means to reimplement it specifically for a given subtype. This facilitates the dynamic dispatch of methods (also known as polymorphism), where an application will execute the most specific method depending on the type of the structured type instance (for example, where it is situated in the structured type hierarchy).

To define an overriding method, use the CREATE TYPE (or ALTER TYPE) statement, and specify the OVERRIDING clause before the METHOD clause. If OVERRIDING is not specified, the original method (belonging to the supertype) will be used. For an overriding method to be defined, the following conditions must be met:

- The type you are creating (or altering) must be a subtype of the structured type whose method you intend to override.
- The signature (the method's name and parameter list) of the method you are declaring is identical to that of a method belonging to the supertype.
- An overriding method must implicitly override exactly one original method.
- The routine you intend to override is a user-defined structured type instance method.
- The original method is not declared with PARAMETER STYLE JAVA.

The following example demonstrates a sample scenario for the overriding of methods:

Data types:

```
CREATE TYPE a AS (z varchar(20))
    METHOD foo(i integer) RETURNS varchar(80)
        LANGUAGE SQL;

CREATE TYPE b UNDER a AS (y varchar(20))
    OVERRIDING METHOD foo(i integer) RETURNS varchar(80);

CREATE TYPE c UNDER a AS (x varchar(20))
    OVERRIDING METHOD foo(i integer) RETURNS varchar(80);

CREATE TYPE d UNDER b AS (w varchar(20))
    OVERRIDING METHOD foo(i integer) RETURNS varchar(80);
```

In this situation, a is the supertype. Types b and c are subtypes of a. Finally, d is the subtype of b

Methods:

```
CREATE METHOD foo(i integer) FOR a
  RETURN "In method foo_a. Input: " | char(i) | self..z | ".";

CREATE METHOD foo(i integer) FOR b
  RETURN "In method foo_b. Input: " | char(i) | self..z |
         " y = " | self..y | ".";

CREATE METHOD foo(i integer) FOR c
  RETURN "In method foo_c. Input: " | char(i) | self..z |
         " y = " | self..y | " x = " | self..x | ".";
```

```
CREATE METHOD foo(i integer) FOR d
  RETURN "In method foo_d. Input: " | char(i) | self..z |
         " y = " | self..y | " w = " | self..w | ".";
```

The original method here is fooA. fooB, fooC, and fooD explicitly override fooA. fooD implicitly overrides fooB and fooA. Similarly, fooB implicitly overrides fooA, and fooC implicitly overrides fooA. (Note that explicit overriding implies implicit overriding.)

Related concepts:

- "User-Defined Structured Types" on page 200
- "Structured Type Hierarchies" on page 203

Related tasks:

- "Defining Structured Types" on page 201
- "Defining Behavior for Structured Types" on page 206

Related reference:

- "ALTER TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*

System-Generated Routines for Structured Types

Comparison and Casting Functions for Structured 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 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 automatically creates functions with the following format:

```
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 >=.

Related concepts:

- "User-Defined Structured Types" on page 200
- "Reference Types" on page 223

Related tasks:

- "Defining Structured Types" on page 201

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

Constructor Functions for Structured Types

When you create a structured type, DB2® creates 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 structured type `Person_t`, for example, has the following format:

```
CREATE FUNCTION Person_t ( ) RETURNS Person_t
```

For the subtype `Manager_t`, a constructor with the following format is created:

```
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.

Related concepts:

- "User-Defined Structured Types" on page 200

Related tasks:

- "Defining Structured Types" on page 201

Mutator Methods for Structured Types

A mutator method exists for each attribute of an object. The instance of a structured 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, has the following format:

```
ALTER TYPE Person_t
   ADD METHOD AGE(int)
   RETURNS Person_t;
```

Related concepts:

- “User-Defined Structured Types” on page 200

Related tasks:

- “Defining Structured Types” on page 201

Observer Methods for Structured Types

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, has the following format:

```
ALTER TYPE Person_t
   ADD METHOD AGE()
   RETURNS INTEGER;
```

To invoke a method on a structured type, use the method invocation operator: ‘..’.

The following example demonstrates the use of observer methods for the Person_t type:

```
CREATE FUNCTION MailingAddress (p Person_t)
  RETURNS VARCHAR(40)
  RETURN p..name() || ' ' || p..address()
```

In this function, the name column and address column from a Person_t instance are retrieved via their observer methods and concatenated into a single string to form a mailing address.

Related concepts:

- “User-Defined Structured Types” on page 200

Related tasks:

- “Defining Structured Types” on page 201

Typed Tables

Typed Tables

Typed tables are tables that are defined with a user-defined structured type. With typed tables, you can establish a hierarchical structure with a defined relationship between those tables called a table hierarchy. The table hierarchy is made up of a single root table, supertables, and subtables.

Typed tables store instances of structured types as rows, in which each attribute of the type is stored in a separate column.

Related concepts:
- "User-Defined Structured Types" on page 200
- "Reference Types" on page 223
- "Substitutability in Typed Tables" on page 217
- "Typed Views" on page 228

Related tasks:
- "Storing Objects in Typed Table Rows" on page 218
- "Dropping Typed Tables" on page 216
- "Defining System-Generated Object Identifiers" on page 220
- "Defining Constraints on Object Identifier Columns" on page 222
- "Creating Typed Tables" on page 212

Related reference:
- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "DROP statement" in the *SQL Reference, Volume 2*

Creating Typed Tables

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 typed tables, you must create a corresponding table hierarchy.

The figure below illustrates a typed table hierarchy. The example that follows the figure illustrates the creation of this hierarchy.

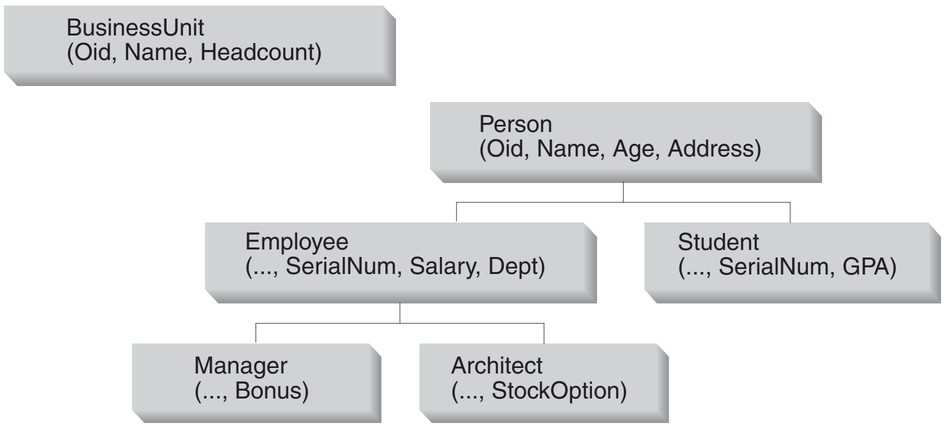


Figure 4. Typed table hierarchy

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. It is common practice in object-oriented design to completely separate the data from the object identifier. For that reason, you cannot update the value of the object identifier after you insert the object identifier. If you want DB2 to generate the OID values,you can use a a SEQUENCE or the GENERATE_UNIQUE() function.

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 by default 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.

Indicating That SELECT Privileges Are Inherited

The mandatory 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.

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;
```

Related concepts:

- "User-Defined Structured Types" on page 200

- “Structured Type Hierarchies” on page 203
- “Typed Tables” on page 211

Related tasks:
- “Defining Structured Types” on page 201
- “Storing Objects in Typed Table Rows” on page 218
- “Dropping Typed Tables” on page 216
- “Defining System-Generated Object Identifiers” on page 220
- “Defining Constraints on Object Identifier Columns” on page 222

Related reference:
- “CREATE TABLE statement” in the *SQL Reference, Volume 2*
- “CREATE TYPE (Structured) statement” in the *SQL Reference, Volume 2*

Dropping Typed Tables

Dropping a typed table is similar to dropping a non-typed table. An important difference is that you must ensure that the table you are dropping has no subtables. If the table you are trying to drop does have subtables, an error will occur. The following example shows how to drop the `Architect` table:

```
DROP TABLE Architect;
```

When a subtable is dropped from a table hierarchy, the columns associated with the subtable are no longer accessible. Through substitutability, dropping a subtable has the semantic effect of deleting all the rows of the subtable from the supertables. This may result in the activation of triggers or referential integrity constraints defined on the supertables.

Other database objects such as tables and indexes will not be affected although packages and cached dynamic statements are marked invalid.

You can also drop an entire table hierarchy. Simply add the HIERARCHY clause to the DROP TABLE statement and name the root table of the hierarchy. For example:

```
DROP TABLE HIERARCHY Person;
```

Dropping a table hierarchy will not result in the activation of triggers or referential integrity contraints.

Related concepts:
- “Structured Type Hierarchies” on page 203
- “Typed Tables” on page 211

Related reference:

- "DROP statement" in the *SQL Reference, Volume 2*

Substitutability in Typed Tables

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 structured type `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. However, this does not mean that the UPDATE statement can change the type of a row if, for instance, a Person_t row is to be updated with Employee_t data. For this to work, the Person_t row would have to be deleted, and the Employee_t row inserted as a new type.

To restrict substitutability in SELECT, UPDATE, or DELETE statements, you can use the ONLY clause. For example, UPDATE ONLY(Person) SET will update rows only in the Person table and not in its subtables.

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 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`. External table functions cannot return structured type columns.

Because a column or table is defined with one type but might contain instances of subtypes, 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.

Related concepts:

- "Structured Type Hierarchies" on page 203

- “Typed Tables” on page 211

Related tasks:

- “Creating a Structured Type Hierarchy” on page 204
- “Issuing Queries to Dereference References” on page 232

Storing Objects in Typed Table Rows

When storing objects as rows in a table, each column of the table contains one attribute of the object. Just as with non-typed tables, you must provide data for all columns that are defined as NOT NULL, including the object identifier column. Because the object identifier column is a REF type, which is strongly typed, you must cast the user-provided object identifier values using the system-generated cast function (which was created for you when you created the structured type). For example, you can store an instance of a person, in a table that contains a column for name and a column for age. First, 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 can 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 this:

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 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';
```

You can access instances of both employees and people just by executing your SQL statement on the Person table. This feature is called *substitutability*. 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;
```

Related concepts:

- “Relationships between Objects in Typed Tables” on page 224
- “Substitutability in Typed Tables” on page 217
- “Typed Tables” on page 211

Related tasks:
- "Storing Instances of Structured Types" on page 202
- "Creating Typed Tables" on page 212

Defining System-Generated Object Identifiers

There are two common approaches of generating unique values, both of which can be applied to object identifiers:
- with sequences
- with the GENERATE_UNIQUE function

If you need to use numeric values as object identifiers, you can use a sequence. To begin, use the REF USING clause to specify that the base type of the object reference is to be a numeric type, in the following case, an INT:

```
CREATE TYPE BusinessUnit_t AS
  (Name VARCHAR(20),
   Headcount INT)
   REF USING INT
   MODE DB2SQL
```

The typed table definition is as follows:

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
  (REF IS oid USER GENERATED)
```

The sequence to generate object identifiers can be defined as follows:

```
CREATE SEQUENCE BusinessUnitOid AS REF(BusinessUnit_t)
```

Note that modifying data in a subtable implicitly modifies all supertables. Therefore, the trigger that invokes the sequence to generate the object identifier is best added to the root of the table hierarchy.

```
CREATE TRIGGER Gen_Bunit_oid
  NO CASCADE
  BEFORE INSERT ON BusinessUnit
  REFERENCING NEW AS new
  FOR EACH ROW
  MODE DB2SQL
  SET new.oid = NEXTVAL FOR BusinessUnitOid
```

Note that since the sequence is defined as REF(BusinessUnitOid), no casting is required to assign to the oid column.

A new business unit can now be added:

```
INSERT INTO BusinessUnit (Name, Headcount)
  VALUES('Software', 10)
```

The usage of a sequence also enables you to retrieve the generated object identifier and use it in subsequent statements. For example, you can add an employee to the Software BusinessUnit assuming the Dept column is of type REF(BusinessUnit):

```
INSERT INTO Employee(Name, Age, SerialNum, Salary, Dept)
  VALUES('Tom', 28, 106, 60000, PREVVAL FOR BusinessUnitOid)
```

As an alternative to using sequences to generate object identifiers, you can use the GENERATE_UNIQUE function. Because GENERATE_UNIQUE returns a CHAR (13) FOR BIT DATA value, ensure that the 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'));
```

Instead of inserting the generated object identifier explicitly on the INSERT statement, you can encapsulate the generation and insertion of the object identifier in a trigger. A trigger on the root of the hierarchy can automate the invocation of the GENERATE_UNIQUE function. The following trigger will generate identifiers for inserts into the Person, Employee, Architect, and Manager tables.

```
CREATE TRIGGER Gen_Person_oid
  NO CASCADE
  BEFORE INSERT ON Person
```

```
   REFERENCING NEW AS new
   FOR EACH ROW
   MODE DB2SQL
   SET new.oid = Person_t (generate_unique());
```

Related concepts:
- "Reference Types" on page 223
- "Relationships between Objects in Typed Tables" on page 224

Related tasks:
- "Creating a Structured Type Hierarchy" on page 204
- "Issuing Queries to Dereference References" on page 232
- "Creating Typed Tables" on page 212

Related reference:
- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE SEQUENCE statement" in the *SQL Reference, Volume 2*

Defining 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 5.

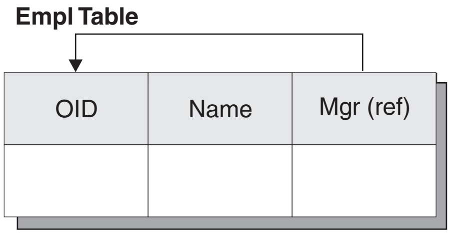


Figure 5. Self-referencing type example

To define constraints on an object identifier column to create a self-referencing relationship on an object:

Step 1. Create the type, for example:

```
CREATE TYPE Empl_t AS
   (Name VARCHAR(10), Mgr REF(Empl_t))
   MODE DB2SQL;
```

Step 2. Create the typed table, for example:

```
CREATE TABLE Empl OF Empl_t
   (REF IS Oid USER GENERATED);
```

Step 3. Add the primary or unique constraint on the Oid column, for example:

```
ALTER TABLE Empl ADD CONSTRAINT pk1 UNIQUE(Oid);
```

Step 4. Add the foreign key constraint, for example:

```
ALTER TABLE Empl ADD CONSTRAINT fk1 FOREIGN KEY(Mgr)
   REFERENCES Empl (Oid);
```

Related concepts:
- "Reference Types" on page 223

Related tasks:
- "Defining Structured Types" on page 201
- "Storing Objects in Typed Table Rows" on page 218
- "Defining System-Generated Object Identifiers" on page 220

Related reference:
- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

Reference Types

Reference 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. You can also use reference types in SQL statements in the same way that you use 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.

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).

Related concepts:

- "Referential Integrity versus Scoped References" on page 228
- "Relationships between Objects in Typed Tables" on page 224
- "Typed Tables" on page 211

Related tasks:

- "Storing Objects in Typed Table Rows" on page 218
- "Issuing Queries to Dereference References" on page 232
- "Defining System-Generated Object Identifiers" on page 220
- "Defining Constraints on Object Identifier Columns" on page 222
- "Creating Typed Tables" on page 212

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 6.

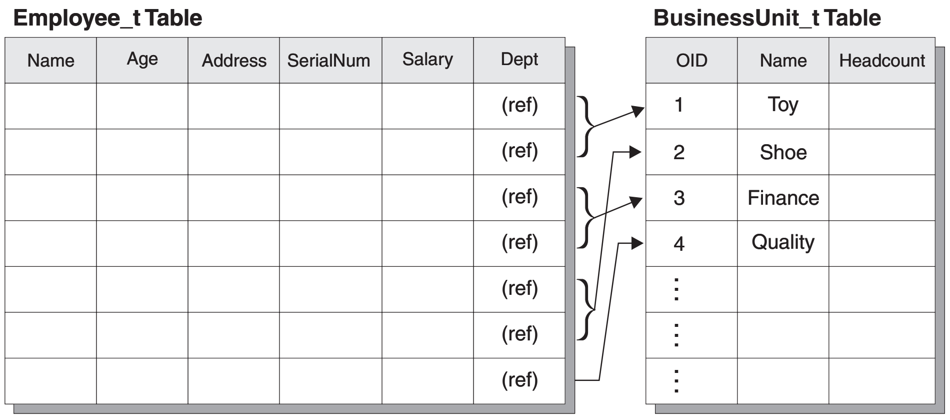


Figure 6. Structured type references from Employee_t to BusinessUnit_t

A normal table (a table that is not a typed table) can have a REF column that refers to a typed table. However, a typed table cannot have a REF column that points to a normal table.

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;
```

Related concepts:

- "Reference Types" on page 223
- "Referential Integrity versus Scoped References" on page 228
- "Typed Tables" on page 211

Related tasks:

- "Restricting Returned Types Using a TYPE Predicate" on page 235
- "Defining System-Generated Object Identifiers" on page 220

Defining Semantic Relationships with References

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. The WITH OPTIONS clause of CREATE TABLE defines the column properties for a column in a typed table. These definable table properties include the relationship between a column in one table and the objects in the same (or another) table. In the example illustrated below, the department for each employee is actually a reference to an object in the `BusinessUnit` table. To define the destination objects of a given reference column, use the SCOPE keyword on the WITH OPTIONS clause.

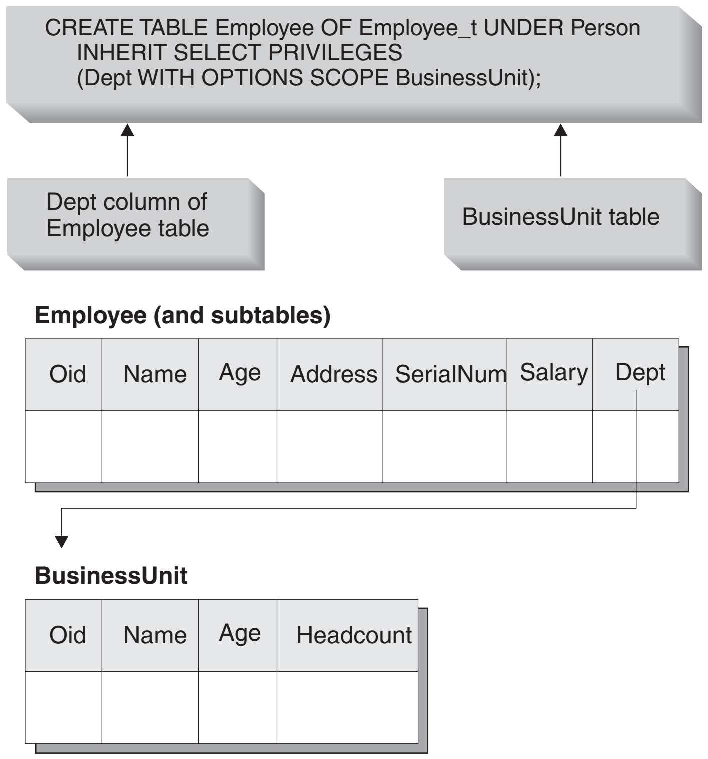

CREATE TABLE Employee OF Employee_t UNDER Person
INHERIT SELECT PRIVILEGES
(Dept WITH OPTIONS SCOPE BusinessUnit);
Dept column of Employee table
BusinessUnit table
Employee (and subtables)
Oid
Name
Age
Address
SerialNum
Salary
Dept
BusinessUnit
Oid
Name
Age
Headcount

Figure 7. 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)
```

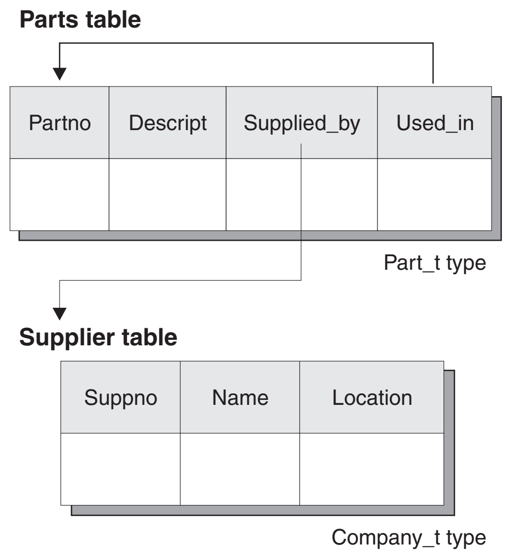


Figure 8. Example of a self-referencing scope

You can use scoped references to write queries that, without scoped references, would have to be written as outer joins or correlated subqueries. For example, the two following queries retrieve the supplier of the part in which the part ’1234’ is being used:

```
SELECT Used_in->Supplied_by->Name
  FROM Parts
  WHERE Partno = Part_t('1234')
```

Without a a scoped reference the query looks like this:

```
SELECT S.Name
  FROM (Parts AS P RIGHT OUTER JOIN Parts C ON P.Used_in = C.Partno)
    RIGHT OUTER JOIN Suppliers S ON C.Supplied_by = S.Suppno
  WHERE P.Partno = Part_t('1234')
```

Related concepts:

- “Reference Types” on page 223
- “Referential Integrity versus Scoped References” on page 228
- “Relationships between Objects in Typed Tables” on page 224
- “Typed Tables” on page 211

Related tasks:

- "Defining System-Generated Object Identifiers" on page 220

Referential Integrity versus 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. To ensure that the objects in these relationships exist, you must add a referential constraint between the tables.

Related concepts:

- "Reference Types" on page 223
- "Typed Tables" on page 211

Related tasks:

- "Defining Semantic Relationships with References" on page 225

Typed Views

Typed Views

For typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed view. Rows of the typed view can be thought of as a representation of instances of the structured type.

Like a typed table, a typed view can be part of a view hierarchy. A subview inherits columns from its superview. The term subview applies to all typed views that are below a typed view in the view hierarchy. A proper subview of a view V is a view below V in the typed view hierarchy.

Related concepts:

- "User-Defined Structured Types" on page 200
- "Typed Tables" on page 211

Related tasks:

- "Creating Typed Views" on page 229
- "Altering Typed Views" on page 231
- "Dropping Typed Views" on page 232

Related reference:

- “ALTER VIEW statement” in the *SQL Reference, Volume 2*
- “CREATE VIEW statement” in the *SQL Reference, Volume 2*
- “DROP statement” in the *SQL Reference, Volume 2*

Creating Typed Views

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 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 root view, you must specify an object identifier column for the view. If you create a subview, it inherits the object identifier column.

The USER GENERATED clause specifies that the 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 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:

```
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. 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.

Related concepts:
- "User-Defined Structured Types" on page 200
- "Typed Tables" on page 211
- "Typed Views" on page 228

Related reference:
- "CREATE VIEW statement" in the *SQL Reference, Volume 2*

Altering Typed Views

The ALTER VIEW statement modifies an existing view by altering a reference type column to add a scope. Any other changes you intend to 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).

Related concepts:
- "User-Defined Structured Types" on page 200
- "Typed Tables" on page 211
- "Typed Views" on page 228

Related tasks:
- "Creating Typed Views" on page 229

Related reference:
- "ALTER VIEW statement" in the *SQL Reference, Volume 2*

Dropping Typed Views

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.

Other database objects such as tables and indexes will not be affected although packages and cached dynamic statements are marked invalid.

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;
```

Related concepts:

- "User-Defined Structured Types" on page 200
- "Typed Tables" on page 211
- "Typed Views" on page 228

Related tasks:

- "Creating Typed Views" on page 229

Related reference:

- "DROP statement" in the *SQL Reference, Volume 2*

Querying Typed Tables and Typed Views

Issuing Queries to 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. 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.

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.

Related concepts:

- "User-Defined Structured Types" on page 200
- "Reference Types" on page 223
- "Relationships between Objects in Typed Tables" on page 224
- "Typed Tables" on page 211
- "Typed Views" on page 228

Related tasks:

- "Storing Objects in Typed Table Rows" on page 218
- "Returning Objects of a Particular Type Using ONLY" on page 234
- "Restricting Returned Types Using a TYPE Predicate" on page 235
- "Returning All Possible Types Using OUTER" on page 236
- "Defining System-Generated Object Identifiers" on page 220

Related reference:

- "DEREF scalar function" in the *SQL Reference, Volume 1*

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.

Related concepts:

- "User-Defined Distinct Types" on page 179
- "Typed Tables" on page 211

Related tasks:

- "Storing Objects in Typed Table Rows" on page 218
- "Issuing Queries to Dereference References" on page 232

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
```

Related concepts:
- "User-Defined Structured Types" on page 200
- "Reference Types" on page 223
- "Typed Tables" on page 211
- "Typed Views" on page 228

Related tasks:
- "Storing Objects in Typed Table Rows" on page 218
- "Issuing Queries to Dereference References" on page 232

Returning All Possible Types 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. 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                 ...
```

Related concepts:

- “User-Defined Structured Types” on page 200
- “Typed Tables” on page 211
- “Typed Views” on page 228

Related tasks:

- “Storing Structured Type Objects in Table Columns” on page 237
- “Issuing Queries to Dereference References” on page 232

Structured Types as Column Types

Storing Structured Type Objects in Table Columns

Storing objects in columns is useful when you need to model facts about your business objects that cannot be adequately modeled 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 modeled 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 9.

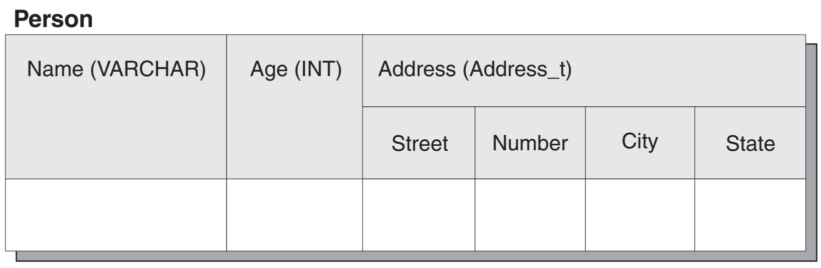


Figure 9. Address attribute as a structured type

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.

Figure 10 shows a hierarchy for the different types of addresses. 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 region.

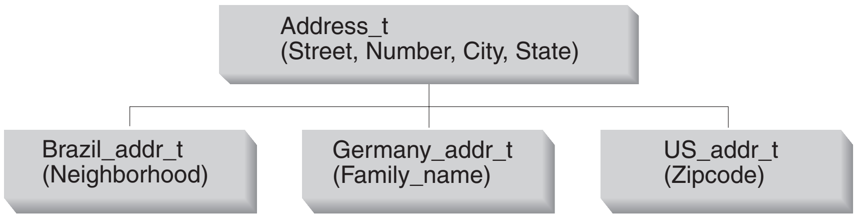


Figure 10. Structured type hierarchy for Address_t type

```
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;
```

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 example uses one observer method and two mutator methods, one for the `Number` attribute and one for the `Street` attribute, 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`.

To allow for updating of more complex, especially nested, instances of structured types, DB2 also allows you to drill down to the attribute to be updated on the left-hand side of the SET clause:

```
UPDATE Employee
   SET Address..Number = '4869',
       Address..Street = 'Appletree'
   WHERE Name='Franky' AND Address..State='CA'
```

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.

Related concepts:

- "User-Defined Structured Types" on page 200

Related tasks:

- "Defining Structured Types" on page 201
- "Storing Instances of Structured Types" on page 202
- "Inserting Structured Type Attributes Into Columns" on page 240
- "Retrieving and Modifying Structured Type Values in Columns" on page 243

Related reference:

- "UPDATE statement" in the *SQL Reference, Volume 2*

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)

Related concepts:

- "Observer Methods for Structured Types" on page 211

Related tasks:

- "Defining Structured Types" on page 201
- "Storing Structured Type Objects in Table Columns" on page 237
- "Retrieving Structured Type Attributes" on page 244

Defining and Altering Tables with Structured Type Columns

Creating a table with columns of structured types is for the most part no different than creating tables with only the DB2 SQL data types. For every column that is defined, a corresponding data type is assigned. For structured type columns, the structured type name is provided as the corresponding data type. For example, the following ALTER TABLE 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.

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. This clause will indicate the maximum size of an instance of a structured type in a column. If the size of a structured type instance is less than the defined maximum, the data will be stored inline with the rest of the values in the row. If the size of the structured type exceeds the defined maximum, the structured type data is stored outside of the table (much like LOBs).

To accommodate changes you make to a structured type, you can alter the affected structured type column's size by issuing the ALTER TABLE ALTER COLUMN SET INLINE LENGTH statement. After altering a column's length you should invoke the REORG utility.

Related concepts:

- "User-Defined Structured Types" on page 200

Related tasks:

- "Defining Structured Types" on page 201
- "Storing Structured Type Objects in Table Columns" on page 237

Related reference:

- "ALTER TABLE statement" in the *SQL Reference, Volume 2*
- "CREATE TABLE statement" in the *SQL Reference, Volume 2*

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.

Related tasks:

- "Defining Structured Types" on page 201
- "Storing Structured Type Objects in Table Columns" on page 237

Related reference:

- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*

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 to 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 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 US_addr_t()..street(street)..number(number)
       ..city(city)..state(state)..zip(zipcode);
```

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'));
```

Related concepts:

- "Substitutability in Typed Tables" on page 217
- "Typed Tables" on page 211

Related tasks:

- "Defining Structured Types" on page 201
- "Storing Structured Type Objects in Table Columns" on page 237
- "Inserting Structured Type Attributes Into Columns" on page 240
- "Defining and Altering Tables with Structured Type Columns" on page 240
- "Defining Types with Structured Type Attributes" on page 241

Modifying Structured Type Values in Columns

Retrieving and Modifying Structured Type Values in Columns

There are two ways that applications and user-defined functions can access data in structured type columns: by accessing individual attributes of an object, or by assessing the object as a single value. If you want to treat an object as a single value, you must first define transform functions. 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;
```

The following topics describe how you can explicitly access individual attributes of an object by invoking the DB2 built-in observer and mutator methods. These built-in methods do not require you to define a transform function.

Procedure:

1. Retrieving Structured Type Attributes
2. Accessing the Attributes of Subtypes
3. Modifying Structured Type Attributes
4. Returning Information About a Structured Type

Related concepts:

- "Transform Functions and Transform Groups" on page 246

Related tasks:

- "Retrieving Structured Type Attributes" on page 244
- "Accessing the Attributes of Subtypes" on page 245
- "Modifying Structured Type Attributes" on page 245
- "Returning Information About a Structured Type" on page 246
- "Storing Structured Type Objects in Table Columns" on page 237
- "Inserting Structured Type Attributes Into Columns" on page 240
- "Inserting Rows That Contain Structured Type Values" on page 242

Retrieving Structured Type 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';
```

Related tasks:

- "Inserting Structured Type Attributes Into Columns" on page 240
- "Accessing the Attributes of Subtypes" on page 245
- "Modifying Structured Type Attributes" on page 245
- "Returning Information About a Structured Type" on page 246

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`. 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;
```

Related tasks:

- "Inserting Structured Type Attributes Into Columns" on page 240
- "Retrieving Structured Type Attributes" on page 244
- "Modifying Structured Type Attributes" on page 245
- "Returning Information About a Structured Type" on page 246

Modifying Structured Type 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';
```

Related tasks:

- "Inserting Structured Type Attributes Into Columns" on page 240
- "Retrieving Structured Type Attributes" on page 244
- "Accessing the Attributes of Subtypes" on page 245
- "Returning Information About a Structured Type" on page 246

Returning Information About a Structured Type

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';
```

Related tasks:

- "Inserting Structured Type Attributes Into Columns" on page 240
- "Retrieving Structured Type Attributes" on page 244
- "Accessing the Attributes of Subtypes" on page 245
- "Modifying Structured Type Attributes" on page 245

Transform Functions and Transform Groups

Transform Functions and Transform Groups

Transform functions are used to exchange structured type values with host language programs and with external functions and methods. 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.

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;
```

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.

Related concepts:

- “User-Defined Structured Types” on page 200
- “Transform Function Requirements” on page 264
- “Specification of Transform Groups” on page 249
- “Host Language Program Mappings with Transform Functions” on page 252
- “Function Transforms” on page 253
- “Recommendations for Naming Transform Groups” on page 248

Related tasks:

- “Retrieving and Modifying Structured Type Values in Columns” on page 243

Related reference:

- “DROP statement” in the *SQL Reference, Volume 2*
- “CREATE TRANSFORM statement” in the *SQL Reference, Volume 2*

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, 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, DB2® invokes the correct transform function whenever you bind in or bind out an address or polygon.

Restriction: You cannot 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.

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Specification of Transform Groups" on page 249

Related reference:

- "CREATE TRANSFORM statement" in the *SQL Reference, Volume 2*

Specification of Transform Groups

Specification of Transform Groups

Many transform groups can be defined for a given structured type, so 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.
- 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.
- When executing dynamic SQL, or when using the Command Line Processor, you must specify the group of transforms that perform client bind in and bind out for a referenced type.

Related concepts:

- "Transform Functions and Transform Groups" on page 246

- “Host Language Program Mappings with Transform Functions” on page 252

Related tasks:

- “Specifying Transform Groups for External Routines” on page 250
- “Specifying Transform Groups for Dynamic SQL” on page 250
- “Specifying Transform Groups for Static SQL” on page 251

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 ;
```

Related concepts:

- “Transform Functions and Transform Groups” on page 246
- “Specification of Transform Groups” on page 249

Related tasks:

- “Defining Behavior for Structured Types” on page 206
- “Specifying Transform Groups for Dynamic SQL” on page 250
- “Specifying Transform Groups for Static SQL” on page 251

Specifying Transform Groups 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;
```

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Specification of Transform Groups" on page 249

Related tasks:

- "Specifying Transform Groups for External Routines" on page 250
- "Specifying Transform Groups for Static SQL" on page 251

Specifying Transform Groups 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
... ;
```

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Specification of Transform Groups" on page 249

Related tasks:

- "Specifying Transform Groups for External Routines" on page 250
- "Specifying Transform Groups for Dynamic SQL" on page 250

Related reference:

- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

Creating the Mapping to the Host Language Program

Host Language Program Mappings with Transform Functions

An application cannot directly select an entire object, although 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.

Related concepts:

- "Transform Function Requirements" on page 264
- "Transform Functions and Transform Groups" on page 246
- "Function Transforms" on page 253
- "Client Transforms" on page 259

Related tasks:

- "Implementing Function Transforms Using SQL-bodied Routines" on page 255
- "Passing Structured Type Parameters to External Routines" on page 257

Related reference:

- "CREATE TRANSFORM statement" in the *SQL Reference, Volume 2*

Function Transforms

DB2® uses 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, 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 11 on page 254 shows how DB2 processes the address.

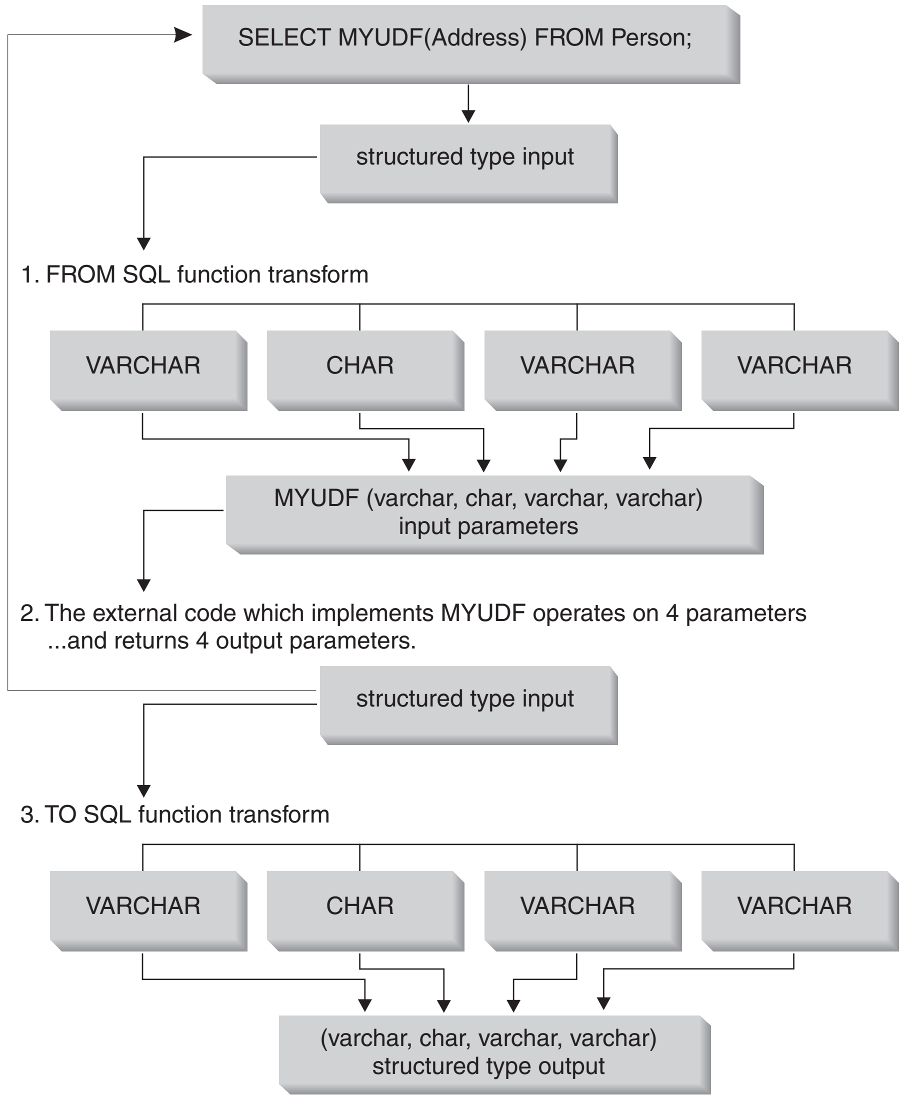

SELECT MYUDF(Address) FROM Person;
structured type input
1. FROM SQL function transform
VARCHAR
CHAR
VARCHAR
VARCHAR
MYUDF (varchar, char, varchar, varchar) input parameters
2. The external code which implements MYUDF operates on 4 parameters ...and returns 4 output parameters.
structured type input
3. TO SQL function transform
VARCHAR
CHAR
VARCHAR
VARCHAR
(varchar, char, varchar, varchar) structured type output

Figure 11. 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.

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%';
```

Related concepts:

- “Transform Functions and Transform Groups” on page 246
- “Host Language Program Mappings with Transform Functions” on page 252
- “Client Transforms” on page 259

Related tasks:

- “Implementing Function Transforms Using SQL-bodied Routines” on page 255
- “Passing Structured Type Parameters to External Routines” on page 257

Implementing Function Transforms Using SQL-bodied Routines

To decompose and construct objects when exchanging the object with an external routine, you must use user-defined functions written in SQL, called SQL-bodied functions. To create a SQL-bodied function, 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. 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 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.

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 using this transform function must use this same order.

Related concepts:

- "Function Transforms" on page 253

Related reference:

- "CREATE FUNCTION (SQL Scalar, Table or Row) statement" in the *SQL Reference, Volume 2*

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 that implement the UDFs:

```
void SQL_API_FN stream_to_client(
/* decomposed address */
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
/* VARCHAR output */
   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)
```

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
	st2_att3 INTEGER	

The following code represents the C language headers for routines that 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,
  /* Decomposed st1 input */
     SQLUDF_VARCHAR *st1_att1,
     SQLUDF_INTEGER  *st1_att2,
  /* Decomposed st2 input */
     SQLUDF_VARCHAR *st2_att1,
     SQLUDF_CHAR   *st2_att2,
     SQLUDF_INTEGER *st2_att3,
  /* Decomposed 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
  )
```

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Host Language Program Mappings with Transform Functions" on page 252

- “Function Transforms” on page 253
- “Client Transforms” on page 259

Related tasks:

- “Implementing Client Transforms for Binding in from a Client Using External UDFs” on page 263

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 12 on page 260 shows the process of binding out that address to the client program.

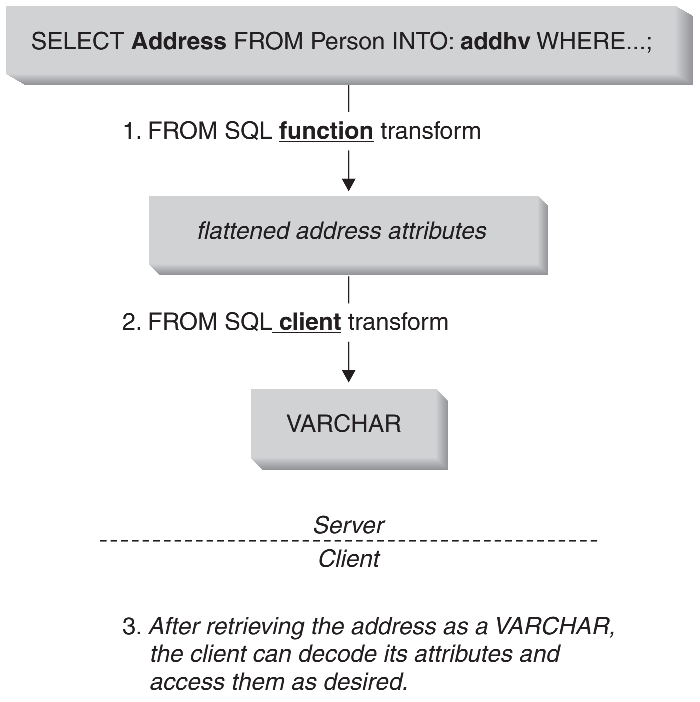

SELECT Address FROM Person INTO: addhv WHERE...;
1. FROM SQL function transform
flattened address attributes
2. FROM SQL client transform
VARCHAR
Server
Client
3. After retrieving the address as a VARCHAR, the client can decode its attributes and access them as desired.

Figure 12. Binding out a structured type to a client application

1. The object must first be passed to the FROM SQL function transform to decompose the object 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.
3. The client program processes the value.

Figure 13 on page 261 shows the reverse process of passing the address back to the database.

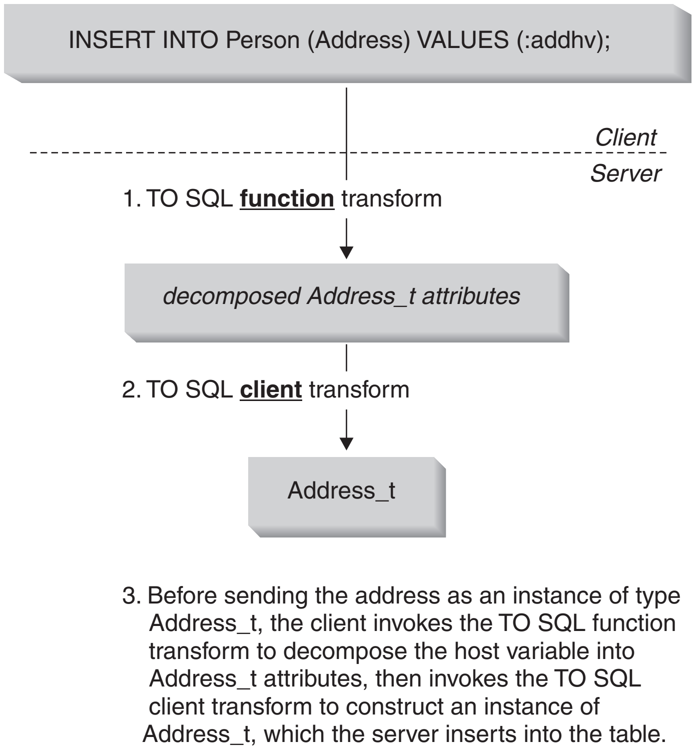

INSERT INTO Person (Address) VALUES (:addhv);
Client
Server
1. TO SQL function transform
decomposed Address_t attributes
2. TO SQL client transform
Address_t
3. Before sending the address as an instance of type Address_t, the client invokes the TO SQL function transform to decompose the host variable into Address_t attributes, then invokes the TO SQL client transform to construct an instance of Address_t, which the server inserts into the table.

Figure 13. 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.

Include the TRANSFORM GROUP clause to tell DB2® which set of transforms to use in processing the address type in the given function.

Related concepts:

- "Host Language Program Mappings with Transform Functions" on page 252
- "Function Transforms" on page 253

Related tasks:

- "Implementing Client Transforms Using External UDFs" on page 262

- “Implementing Client Transforms for Binding in from a Client Using External UDFs” on page 263

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 SQL;
```

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;
```

Notice that the DDL in `from_sql_to_client` includes a clause called TRANSFORM GROUP. This clause tells DB2 which set of transforms to use in processing the address type in those functions.

Related concepts:

- “Client Transforms” on page 259

Related tasks:

- “Passing Structured Type Parameters to External Routines” on page 257
- “Implementing Client Transforms for Binding in from a Client Using External UDFs” on page 263

Implementing Client Transforms for Binding in from a Client Using External UDFs

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 SQL;
```

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.

Notice that the DDL in `to_sql_from_client` includes a clause called TRANSFORM GROUP. This clause tells DB2 which set of transforms to use in processing the address type in those functions.

Related concepts:

- "Client Transforms" on page 259

Related tasks:

- "Implementing Client Transforms Using External UDFs" on page 262

Data Conversion Considerations

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.

As much as possible, 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.

Related concepts:

- "Transform Functions and Transform Groups" on page 246
- "Host Language Program Mappings with Transform Functions" on page 252
- "Function Transforms" on page 253

Transform Function Requirements

Table 15 on page 265 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.

Related concepts:

- "Transform Functions and Transform Groups" on page 246

Related tasks:

- "Retrieving Subtype Data from DB2" on page 266

Retrieving Subtype Data from DB2

If your data model takes advantage of subtypes, a value in a column could be one of many different subtypes. 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 an 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:

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 Step 6.

The transform group contains the transform function Addr_stream associated with the root type Address_t in Step 5 on page 268. Addr_stream is a SQL-bodied function, defined in Step 4 on page 268, 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 Step 3 on page 266. 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 Step 3 on page 266 name the TRANSFORM GROUP that contains these transforms.

The CREATE FUNCTION statements for the transform functions are issued in Step 1 on page 266. The CREATE TRANSFORM statements that associate the transform functions with their transform groups are issued in Step 2 on page 266.

Related concepts:
- "Transform Function Requirements" on page 264
- "Transform Functions and Transform Groups" on page 246

Related reference:
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

Returning Subtype Data to DB2

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:

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 Step 6. DB2 finds the transform function it needs: `stream_address`, which is associated with the root type `Address_t` in Step 5.

`stream_address` is a SQL-bodied function, defined in Step 4 on page 272, 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 Step 3 on page 270. 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 Step 3 on page 270 name the TRANSFORM GROUP that contains the transforms.

The SQL-bodied transform functions from Step 1 on page 269, and the associations with the transform groups from Step 2 on page 270, are named in the CREATE FUNCTION statements of Step 3 on page 270.

Related concepts:

- "Transform Function Requirements" on page 264
- "Transform Functions and Transform Groups" on page 246

Related reference:

- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

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;
```

Related concepts:

- "Transform Functions and Transform Groups" on page 246

Related tasks:

- "Describing a Structured Type" on page 274

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.

Related concepts:

- "Transform Functions and Transform Groups" on page 246

Related tasks:

- "Declaring Structured Type Host Variables" on page 273

Chapter 9. Triggers

Triggers in Application Development

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 a modify operation (insert, update, or delete) 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 modify 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. 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 (from the 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. To implement functionally dependent columns DB2 also supports GENERATED columns. These are columns whose values are always derived in a deterministic fashion from other values in the same row.
- Read from other tables for cross-referencing purposes.
- Write to other tables for audit-trail purposes.
- Support *alerts* (for example, through electronic mail messages).

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.

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.

Related concepts:

- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Trigger Activation Time" on page 284
- "Trigger Interactions with Referential Constraints" on page 279
- "Trigger Creation Guidelines" on page 281
- "INSTEAD OF Triggers" on page 279

Related tasks:

- "Creating Triggers" on page 282
- "Defining Business Rules Using Triggers" on page 297

Related reference:

- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Related samples:

- "tbtrig.out -- HOW TO USE TRIGGERS (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "tbtrig.out -- HOW TO USE TRIGGERS (C++)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "trigsql.sqb -- How to use a trigger on a table (IBM COBOL)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "TbTrig.out -- HOW TO USE TRIGGERS (JDBC)"
- "TbTrig.out -- HOW TO USE TRIGGERS (SQLJ)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"

INSERT, UPDATE, and DELETE Triggers

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 modify operation. The modify 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
```

Related concepts:
- "Trigger Granularity" on page 283
- "Trigger Activation Time" on page 284
- "Trigger Interactions with Referential Constraints" on page 279
- "Triggers in Application Development" on page 275
- "INSTEAD OF Triggers" on page 279

Related tasks:
- "Creating Triggers" on page 282

Trigger Interactions with Referential Constraints

A trigger event can occur as a 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.

Related concepts:
- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Triggers in Application Development" on page 275

INSTEAD OF Triggers

INSTEAD OF triggers describe how to perform insert, update, and delete operations against views that are too complex to support these operations natively. INSTEAD OF triggers allow applications to use a view as the sole interface for all SQL operations (insert, delete, update and select). Usually, INSTEAD OF triggers contain the inverse of the logic applied in a view body. For example, consider a view that decrypts columns from its source table. The INSTEAD OF trigger for this view encrypts data and then inserts it into the source table, thus performing the symmetrical operation.

Using an INSTEAD OF trigger, the requested modify operation against the view gets replaced by the trigger logic, which performs the operation on behalf of the view. From the perspective of the application this happens transparently, as it perceives that all operations are performed against the view. Only one INSTEAD OF trigger is allowed for each kind of operation on a given subject view.

The view itself must be an untyped view or an alias that resolves to an untyped view. Also, it cannot be a view that is defined using WITH CHECK OPTION (a symmetric view) or a view on which a symmetric view has been defined directly or indirectly.

The following example presents three INSTEAD OF triggers that provide logic for INSERTs, UPDATEs, and DELETEs to the defined view (EMPV). The view EMPV contains a join in its from clause and therefore cannot natively support any modify operations.

```
CREATE VIEW EMPV(EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO,
                 HIREDATE, DEPTNAME)
AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO,
          HIREDATE, DEPTNAME
          FROM EMPLOYEE, DEPARTMENT WHERE
               EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO

CREATE TRIGGER EMPV_INSERT INSTEAD OF INSERT ON EMPV
REFERENCING NEW AS NEWEMP DEFAULTS NULL FOR EACH ROW MODE DB2SQL
INSERT INTO EMPLOYEE (EMPNO, FIRSTNME, MIDINIT, LASTNAME,
                      WORKDEPT, PHONENO, HIREDATE)
        VALUES(EMPNO, FIRSTNME, MIDINIT, LASTNAME,
               COALESCE((SELECT DEPTNO FROM DEPARTMENT AS D
                         WHERE D.DEPTNAME = NEWEMP.DEPTNAME),
                        RAISE_ERROR('70001', 'Unknown dept name')),
               PHONENO, HIREDATE)

CREATE TRIGGER EMPV_UPDATE INSTEAD OF UPDATE ON EMPV
REFERENCING NEW AS NEWEMP OLD AS OLDEMP DEFAULTS NULL
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
 VALUES(CASE WHEN NEWEMP.EMPNO = OLDEMP.EMPNO THEN 0
             ELSE RAISE_ERROR('70002', 'Must not change EMPNO') END);
 UPDATE EMPLOYEE AS E
   SET (FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, HIREDATE)
     = (NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME,
               COALESCE((SELECT DEPTNO FROM DEPARTMENT AS D
                         WHERE D.DEPTNAME = NEWEMP.DEPTNAME),
                        RAISE_ERROR ('70001', 'Unknown dept name')),
               NEWEMP.PHONENO, NEWEMP.HIREDATE)
 WHERE NEWEMP.EMPNO = E.EMPNO;
END

CREATE TRIGGER EMPV_DELETE INSTEAD OF DELETE ON EMPV
REFERENCING OLD AS OLDEMP FOR EACH ROW MODE DB2SQL
DELETE FROM EMPLOYEE AS E WHERE E.EMPNO = OLDEMP.EMPNO
```

Related concepts:

- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Triggers in Application Development" on page 275

Related tasks:

- "Creating Triggers" on page 282

Related reference:

- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Trigger Creation Guidelines

When creating a trigger, you must associate it with a table. This table is called the *subject table* of the trigger. The term *modify operation* refers to any change in the state of the subject table. A modify 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 modify 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 modify operation affects and executes the trigger.

When creating a trigger, you must 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 modify 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.

Related concepts:
- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Trigger Activation Time" on page 284
- "Triggers in Application Development" on page 275

Related tasks:
- "Creating Triggers" on page 282

Related reference:
- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Creating Triggers

To create a trigger from the Control Center, use the Create Trigger dialogue. The Create Trigger dialogue can be found by expanding the object tree and right-clicking the Triggers folder.

To create a trigger using the command line, use the following template of the CREATE TRIGGER statement:

```
CREATE TRIGGER <name>
 <action> ON <table_name>
 <operation>
 <triggered_action>
```

The following SQL statement creates a trigger that increases the number of employees each time a new person is hired, by adding 1 to the number of employees (NBEMP) column in the COMPANY_STATS table each time a row is added to the EMPLOYEE table.

```
CREATE TRIGGER NEW_HIRED
 AFTER INSERT ON EMPLOYEE
 FOR EACH ROW MODE DB2SQL
 UPDATE COMPANY_STATS SET NBEMP = NBEMP+1;
```

Related concepts:
- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Trigger Activation Time" on page 284

- "Triggers in Application Development" on page 275
- "Trigger Creation Guidelines" on page 281

Related reference:

- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Related samples:

- "tbtrig.out -- HOW TO USE TRIGGERS (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "tbtrig.out -- HOW TO USE TRIGGERS (C++)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "trigsql.sqb -- How to use a trigger on a table (IBM COBOL)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "TbTrig.out -- HOW TO USE TRIGGERS (JDBC)"
- "TbTrig.out -- HOW TO USE TRIGGERS (SQLJ)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"

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. If you need to refer to the specific rows affected by the triggered action, use FOR EACH ROW granularity. An example of this is the comparison of the new and old values of an updated row in an AFTER UPDATE trigger.

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.

Related concepts:

- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Activation Time" on page 284
- "Triggers in Application Development" on page 275
- "Trigger Creation Guidelines" on page 281

Related tasks:

- "Creating Triggers" on page 282

Trigger Activation Time

The *trigger activation time* specifies when the trigger should be activated. That is, either BEFORE, AFTER, or INSTEAD OF 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. Hence, the subject table will only be modified after the BEFORE trigger has completed execution for each row. 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.

If the activation time is INSTEAD OF, the triggered actions for each row in the set of affected rows are activated instead of executing the trigger event.

INSTEAD OF triggers must have a granularity of FOR EACH ROW, and the subject table must be a view. No other triggers are able to use a view as the subject table.

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 modify 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.

In contrast, you can view an INSTEAD OF trigger as a description of the inverse operation of the view it is defined on. For example, if the select list in the view contains an expression over a base table, the INSERT statement in the body of its INSTEAD OF INSERT trigger will contain the reverse expression.

Because of the different nature of BEFORE, AFTER, and INSTEAD OF triggers, a different set of SQL operations can be used to define the triggered actions of BEFORE and AFTER, INSTEAD OF 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. Similarly, different trigger granularities are supported in BEFORE, AFTER, and INSTEAD OF triggers. For example, the 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 the operation's failure.

The triggered SQL statement of all triggers may be a dynamic compound statement. However, BEFORE triggers face some restrictions; they may not contain the following SQL statements:

- UPDATE
- DELETE
- INSERT

Related concepts:

- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Triggered Action: SQL Statements" on page 291
- "Triggers in Application Development" on page 275
- "Trigger Creation Guidelines" on page 281

Related tasks:

- "Creating Triggers" on page 282

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 AS 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 AS 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.

Related concepts:
- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Transition Tables" on page 288

Related tasks:
- "Creating Triggers" on page 282

Related reference:
- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

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 AS 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 AS 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.

Related concepts:
- "INSERT, UPDATE, and DELETE Triggers" on page 278
- "Trigger Granularity" on page 283
- "Transition Variables" on page 286

Related tasks:
- "Creating Triggers" on page 282

Related reference:
- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Triggered Action

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
```

Related concepts:

- "Triggered Action: Conditions" on page 290
- "Triggered Action: SQL Statements" on page 291
- "Triggered Action: Functions" on page 292

Triggered Action: Conditions

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.

Related concepts:

- "Triggered Action" on page 289
- "Triggered Action: SQL Statements" on page 291

- “Triggered Action: Functions” on page 292

Triggered Action: SQL Statements

The set of triggered SQL statements carries out the real actions caused by activating a trigger. 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.

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).

The triggered SQL statement of all triggers may be a dynamic compound statement. That is, they may contain one or more of the following:

- DECLARE variable statement
- SET variable statement
- WHILE loop
- FOR loop
- IF statement
- SIGNAL statement
- ITERATE statement
- LEAVE statement
- GET DIGNOSTIC statement
- fullselect

However, only AFTER and INSTEAD of triggers may contain one or more of the following:

- UPDATE SQL statement
- DELETE SQL statement
- INSERT SQL statement

Related concepts:

- “Triggered Action” on page 289
- “Triggered Action: Conditions” on page 290
- “Triggered Action: Functions” on page 292

Triggered Action: Functions

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 following:

- the SQL path at the time of creation of the trigger.
- the EXECUTE privilege of the definer of the trigger on the functions in the SQL path.

When invoking functions with side effects, such as sending an email, ensure that the functions are correctly defined as having EXTERNAL ACTION. Otherwise, DB2® may decide not to execute the UDF if it does not change the SQL semantics of the trigger.

UDFs are written in SQL, Java, C, or C++. This enables complex control of logic flows, error handling and recovery, and access to system and library functions. 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 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 > 600.
      AND NEW_EMP.HIREDATE - CURRENT DATE <eq; 10000.
      THEN SEND_NOTE('persmgr', NEW_EMP.EMPNO, 'late.txt')
    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')
    END;
  END
```

Related concepts:

- "Triggered Action" on page 289
- "Triggered Action: Conditions" on page 290
- "Triggered Action: SQL Statements" on page 291

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 are one or more AFTER or INSTEAD OF 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.

Related concepts:

- "Triggers in Application Development" on page 275

Related tasks:

- "Extracting Information from UDTs, UDFs, and LOBs with Triggers" on page 294
- "Preventing Operations on Tables Using Triggers" on page 296
- "Defining Business Rules Using Triggers" on page 297
- "Defining Actions Using Triggers" on page 297

Related reference:

- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*

Synergy Between Triggers, Constraints, and Routines

Extracting Information from UDTs, UDFs, and LOBs with Triggers

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, define a BEFORE trigger to extract the corresponding information as follows:

```
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, N.RECEIVER, N.SENT_ON, N.SUBJECT)
      = (SELECT SENDER, RECEIVER, SENT_ON, SUBJECT FROM
          TABLE(EMAIL_HEADER(N.MESSAGE)) AS H)
  END
```

This can also be done by adding generated columns to the ELECTRONIC_MAIL table.

```
ALTER TABLE ELECTRONIC_MAIL
  ADD COLUMN SENDER VARCHAR(200) GENERATED ALWAYS
    AS (SENDER(N.MESSAGE))
  ADD COLUMN RECEIVER VARCHAR(200) GENERATED ALWAYS
    AS (RECEIVER(N.MESSAGE))
  ADD COLUMN SENT_ON DATE GENERATED ALWAYS
    AS (SENDING_DATE(N.MESSAGE))
  ADD COLUMN SUBJECT VARCHAR(200) GENERATED ALWAYS
    AS (SUBJECT(N.MESSAGE))
```

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.

Related concepts:

- "Triggered Action" on page 289
- "Triggered Action: Conditions" on page 290
- "Triggered Action: SQL Statements" on page 291
- "Triggered Action: Functions" on page 292
- "Multiple Triggers" on page 293

Related tasks:

- "Preventing Operations on Tables Using Triggers" on page 296

- “Defining Business Rules Using Triggers” on page 297
- “Defining Actions Using Triggers” on page 297

Related reference:

- “CREATE TRIGGER statement” in the *SQL Reference, Volume 2*

Preventing Operations on Tables Using Triggers

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'
      SET MESSAGE_TEXT = ('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.

Related concepts:

- “Multiple Triggers” on page 293
- “Triggers in Application Development” on page 275

Related tasks:

- “Extracting Information from UDTs, UDFs, and LOBs with Triggers” on page 294
- “Defining Business Rules Using Triggers” on page 297
- “Defining Actions Using Triggers” on page 297

Related reference:

- “CREATE TRIGGER statement” in the *SQL Reference, Volume 2*

Defining Business Rules Using Triggers

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
```

Related concepts:

- "Multiple Triggers" on page 293
- "Triggers in Application Development" on page 275

Related tasks:

- "Extracting Information from UDTs, UDFs, and LOBs with Triggers" on page 294
- "Preventing Operations on Tables Using Triggers" on page 296
- "Defining Actions Using Triggers" on page 297

Defining Actions Using Triggers

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
```

Related concepts:

- “Multiple Triggers” on page 293
- “Triggers in Application Development” on page 275

Related tasks:

- “Extracting Information from UDTs, UDFs, and LOBs with Triggers” on page 294
- “Preventing Operations on Tables Using Triggers” on page 296

- “Defining Business Rules Using Triggers” on page 297

Related reference:
- “CREATE TRIGGER statement” in the *SQL Reference, Volume 2*

Part 3. Appendixes

Appendix A. DB2GENERAL Routines

DB2GENERAL Routines

PARAMETER STYLE DB2GENERAL routines are written in Java. Creating DB2GENERAL routines is very similar to creating routines 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 cannot call them from UDFs.

When developing routines in Java, it is strongly recommended that you register them using the PARAMETER STYLE JAVA clause in the CREATE statement. PARAMETER STYLE DB2GENERAL is still available to enable the implementation of the following features in Java™ routines:

- table functions
- scratchpads
- access to the DBINFO structure
- the ability to make a FINAL CALL (and a separate first call) to the function or method

If you have PARAMETER STYLE DB2GENERAL routines that do not use any of the above features, it is recommended that you migrate them to PARAMETER STYLE JAVA for portability.

Related concepts:

- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118
- "Table Function Execution Model for Java" on page 57

Related reference:

- "Java Debug Table DB2DBG.ROUTINE_DEBUG" on page 128
- "JAR File Administration on the Database Server" on page 122
- "Supported SQL Data Types in DB2GENERAL Routines" on page 307

- "Java Classes for DB2GENERAL Routines" on page 309
- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310
- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

DB2GENERAL UDFs

You can create and use UDFs in Java™ just as you would in other languages, with only a few minor differences when compared to C UDFs. After you code the UDF, you register it with the database. You can then refer to it in your applications.

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 Java method name
- *T1* through *T4* are the Java types that correspond to SQL types *t1* through *t4*.
- *a*, *b*, and *c* are variable names for the input arguments.
- *d* is an variable name that represents the output argument.

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 routines 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.

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 routines.

Remember that all the Java class files (or the JARs that contain the classes) that you use to implement a routine must reside in the sqllib/function directory, or in a directory specified in the database manager's CLASSPATH.

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. 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.

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.

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.

Related concepts:
- "DB2GENERAL Routines" on page 303
- "Java Routines" on page 118
- "Table Function Execution Model for Java" on page 57

Related reference:
- "Java Debug Table DB2DBG.ROUTINE_DEBUG" on page 128
- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "Java Classes for DB2GENERAL Routines" on page 309

- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310
- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

Related samples:
- "UDFsqlsv.java -- Provide UDFs to be called by UDFsqlcl.java (JDBC)"
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)"
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.sqlj (SQLj)"

Supported SQL Data Types in DB2GENERAL Routines

When you call PARAMETER STYLE DB2GENERAL routines, DB2 converts SQL types to and from Java types for you. Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 16. DB2 SQL Types and Java Objects

SQL Column Type	Java Data Type
SMALLINT	short
INTEGER	int
BIGINT	long
REAL[1]	float
DOUBLE	double
DECIMAL(p,s)	java.math.BigDecimal
NUMERIC(p,s)	java.math.BigDecimal
CHAR(*n*)	java.lang.String
CHAR(*n*) FOR BIT DATA	COM.ibm.db2.app.Blob
VARCHAR(*n*)	java.lang.String
VARCHAR(*n*) FOR BIT DATA	COM.ibm.db2.app.Blob
LONG VARCHAR	java.lang.String
LONG VARCHAR FOR BIT DATA	COM.ibm.db2.app.Blob
GRAPHIC(*n*)	java.lang.String
VARGRAPHIC(*n*)	String
LONG VARGRAPHIC[2]	String
BLOB(*n*)[2]	COM.ibm.db2.app.Blob
CLOB(*n*)[2]	COM.ibm.db2.app.Clob

Table 16. DB2 SQL Types and Java Objects (continued)

SQL Column Type	Java Data Type
DBCLOB(*n*)[2]	COM.ibm.db2.app.Clob
DATE[3]	String
TIME[3]	String
TIMESTAMP[3]	String

Notes:

1. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
2. 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.
3. 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.

Related concepts:

- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118
- "Table Function Execution Model for Java" on page 57

Related reference:

- "Supported SQL Data Types in Java" on page 123
- "Java Classes for DB2GENERAL Routines" on page 309
- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310

- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

Java Classes for DB2GENERAL Routines

Java Classes for DB2GENERAL Routines

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

Related concepts:

- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118

Related reference:

- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310
- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

DB2GENERAL Java Class: 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.

Related concepts:

- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118

Related reference:

- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "Java Classes for DB2GENERAL Routines" on page 309
- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

DB2GENERAL Java Class: 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 getDBprocid() throws Exception
```

This function returns the routine id of the procedure which directly or indirectly invoked this routine. The routine id matches the ROUTINEID column in SYSCAT.ROUTINES which can be used to retrieve the name of the invoking procedure. If the executing routine is invoked from an application, getDBprocid() returns 0.

```
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.

Related concepts:
- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118

Related reference:
- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "Java Classes for DB2GENERAL Routines" on page 309
- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315
- "DB2GENERAL Java Class: COM.IBM.db2.app.Clob" on page 315

DB2GENERAL Java Class: COM.IBM.db2.app.Lob

This class provides utility routines that create temporary Blob or Clob objects for computation inside routines.

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.

Related concepts:
- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118

Related reference:
- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "Java Classes for DB2GENERAL Routines" on page 309

- “DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc” on page 310
- “DB2GENERAL Java Class: COM.IBM.db2.app.UDF” on page 311
- “DB2GENERAL Java Class: COM.IBM.db2.app.Blob” on page 315
- “DB2GENERAL Java Class: COM.IBM.db2.app.Clob” on page 315

DB2GENERAL Java Class: COM.IBM.db2.app.Blob

An instance of this class is passed by the database to represent a BLOB as routine input, and may be passed back as output. The application may create instances, but only in the context of an executing routine. 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.

Related concepts:

- “DB2GENERAL Routines” on page 303
- “DB2GENERAL UDFs” on page 304
- “Java Routines” on page 118

Related reference:

- “Supported SQL Data Types in DB2GENERAL Routines” on page 307
- “Java Classes for DB2GENERAL Routines” on page 309
- “DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc” on page 310
- “DB2GENERAL Java Class: COM.IBM.db2.app.UDF” on page 311
- “DB2GENERAL Java Class: COM.IBM.db2.app.Lob” on page 314
- “DB2GENERAL Java Class: COM.IBM.db2.app.Clob” on page 315

DB2GENERAL Java Class: COM.IBM.db2.app.Clob

An instance of this class is passed by the database to represent a CLOB or DBCLOB as routine input, and may be passed back as output. The application may create instances, but only in the context of an executing routine. 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.

Related concepts:
- "DB2GENERAL Routines" on page 303
- "DB2GENERAL UDFs" on page 304
- "Java Routines" on page 118

Related reference:
- "Supported SQL Data Types in DB2GENERAL Routines" on page 307
- "Java Classes for DB2GENERAL Routines" on page 309
- "DB2GENERAL Java Class: COM.IBM.db2.app.StoredProc" on page 310
- "DB2GENERAL Java Class: COM.IBM.db2.app.UDF" on page 311
- "DB2GENERAL Java Class: COM.IBM.db2.app.Lob" on page 314
- "DB2GENERAL Java Class: COM.IBM.db2.app.Blob" on page 315

Appendix B. DB2 Universal Database technical information

Overview of DB2 Universal Database technical information

DB2 Universal Database technical information can be obtained in the following formats:

- Books (PDF and hard-copy formats)
- A topic tree (HTML format)
- Help for DB2 tools (HTML format)
- Sample programs (HTML format)
- Command line help
- Tutorials

This section is an overview of the technical information that is provided and how you can access it.

Categories of DB2 technical information

The DB2 technical information is categorized by the following headings:

- Core DB2 information
- Administration information
- Application development information
- Business intelligence information
- DB2 Connect information
- Getting started information
- Tutorial information
- Optional component information
- Release notes

The following tables describe, for each book in the DB2 library, the information needed to order the hard copy, print or view the PDF, or locate the HTML directory for that book. A full description of each of the books in the DB2 library is available from the IBM Publications Center at www.ibm.com/shop/publications/order

The installation directory for the HTML documentation CD differs for each category of information:

```
htmlcdpath/doc/htmlcd/%L/category
```

where:

- *htmlcdpath* is the directory where the HTML CD is installed.
- *%L* is the language identifier. For example, en_US.
- *category* is the category identifier. For example, core for the core DB2 information.

In the PDF file name column in the following tables, the character in the sixth position of the file name indicates the language version of a book. For example, the file name `db2d1e80` identifies the English version of the *Administration Guide: Planning* and the file name `db2d1g80` identifies the German 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**
Arabic	w
Brazilian Portuguese	b
Bulgarian	u
Croatian	9
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
Romanian	8
Russian	r
Simp. Chinese	c
Slovakian	7
Slovenian	l
Spanish	z
Swedish	s
Trad. Chinese	t
Turkish	m

No form number indicates that the book is only available online and does not have a printed version.

Core DB2 information

The information in this category cover DB2 topics that are fundamental to all DB2 users. You will find the information in this category useful whether you are a programmer, a database administrator, or you work with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

The installation directory for this category is doc/htmlcd/*%L*/core.

Table 17. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x80
IBM DB2 Universal Database Glossary	No form number	db2t0x80
IBM DB2 Universal Database Master Index	SC09-4839	db2w0x80
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840	db2m1x80
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841	db2m2x80
IBM DB2 Universal Database What's New	SC09-4848	db2q0x80

Administration information

The information in this category covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

The installation directory for this category is doc/htmlcd/*%L*/admin.

Table 18. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x80
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x80
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x80
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x80

Table 18. Administration information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx80
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax80
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx80
IBM DB2 Universal Database Federated Systems Guide	GC27-1224	db2fpx80
IBM DB2 Universal Database Guide to GUI Tools for Administration and Development	SC09-4851	db2atx80
IBM DB2 Universal Database Replication Guide and Reference	SC27-1121	db2e0x80
IBM DB2 Installing and Administering a Satellite Environment	GC09-4823	db2dsx80
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x80
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x80
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x80

Application development information

The information in this category is of special interest to application developers or programmers working with DB2. You will find information about supported languages and compilers, as well as the documentation required to access DB2 using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLj, and CLI. If you view this information online in HTML you can also access a set of DB2 sample programs in HTML.

The installation directory for this category is doc/htmlcd/%*L*/ad.

Table 19. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx80
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x80
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x80
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx80
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx80

Business intelligence information

The information in this category describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

The installation directory for this category is doc/htmlcd/%*L*/wareh.

Table 20. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Information Catalog Center Administration Guide	SC27-1125	db2dix80
IBM DB2 Warehouse Manager Installation Guide	GC27-1122	db2idx80

DB2 Connect information

The information in this category describes how to access host or iSeries data using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

The installation directory for this category is doc/htmlcd/*%L*/conn.

Table 21. DB2 Connect information

Name	Form number	PDF file name
APPC, CPI-C, and SNA Sense Codes	No form number	db2apx80
IBM Connectivity Supplement	No form number	db2h1x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x80
IBM DB2 Connect User's Guide	SC09-4835	db2c0x80

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

The installation directory for this category is doc/htmlcd/*%L*/start.

Table 22. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832	db2itx80
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx80
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x80
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837	db2iyx80
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x80

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

The installation directory for this category is doc/htmlcd/*%L*/tutr.

Table 23. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux80
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax80
Development Center Tutorial for Video Online using Microsoft Visual Basic	No form number	db2tdx80
Information Catalog Center Tutorial	No form number	db2aix80
Video Central for e-business Tutorial	No form number	db2twx80
Visual Explain Tutorial	No form number	db2tvx80

Optional component information

The information in this category describes how to work with optional DB2 components.

The installation directory for this category is doc/htmlcd/*%L*/opt.

Table 24. Optional component information

Name	Form number	PDF file name
IBM DB2 Life Sciences Data Connect Planning, Installation, and Configuration Guide	GC27-1235	db2lsx80
IBM DB2 Spatial Extender User's Guide and Reference	SC27-1226	db2sbx80
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x80

Table 24. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Net Search Extender Administration and Programming Guide **Note:** HTML for this document is not installed from the HTML documentation CD.	SH12-6740	N/A

Release notes

The release notes provide additional information specific to your product's release and FixPak level. They also provides summaries of the documentation updates incorporated in each release and FixPak.

Table 25. Release notes

Name	Form number	PDF file name	HTML directory
DB2 Release Notes	See note.	See note.	doc/prodcd/*%L*/db2ir where *%L* is the language identifier.
DB2 Connect Release Notes	See note.	See note.	doc/prodcd/*%L*/db2cr where *%L* is the language identifier.
DB2 Installation Notes	Available on product CD-ROM only.	Available on product CD-ROM only.	

Note: The HTML version of the release notes 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/opt/db2_08_01 on AIX
  - /opt/IBM/db2/V8.1 on all other UNIX operating systems
- On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.

Related tasks:

- "Printing DB2 books from PDF files" on page 325

- “Ordering printed DB2 books” on page 326
- “Accessing online help” on page 326
- “Finding product information by accessing the DB2 Information Center from the administration tools” on page 330
- “Viewing technical documentation online directly from the DB2 HTML Documentation CD” on page 331

Printing DB2 books from PDF files

You can print DB2 books from the PDF files on the *DB2 PDF Documentation* CD. Using Adobe Acrobat Reader, you can print either the entire book or a specific range of pages.

Prerequisites:

Ensure that you have Adobe Acrobat Reader. It is available from the Adobe Web site at www.adobe.com

Procedure:

To print a DB2 book from a PDF file:

1. Insert the *DB2 PDF Documentation* CD. On UNIX operating systems, mount the DB2 PDF Documentation CD. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start Adobe Acrobat Reader.
3. Open the PDF file from one of the following locations:
   - On Windows operating systems:

     x:\doc\language directory, where *x* represents the CD-ROM drive letter and *language* represents the two-character territory code that represents your language (for example, EN for English).
   - On UNIX operating systems:

     /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.

Related tasks:

- “Ordering printed DB2 books” on page 326
- “Finding product information by accessing the DB2 Information Center from the administration tools” on page 330
- “Viewing technical documentation online directly from the DB2 HTML Documentation CD” on page 331

Related reference:

- “Overview of DB2 Universal Database technical information” on page 317

Ordering printed DB2 books

Procedure:

To order printed books:

- Contact your IBM authorized dealer or marketing representative. To find a local IBM representative, check the IBM Worldwide Directory of Contacts at www.ibm.com/shop/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.
- Visit the IBM Publications Center at www.ibm.com/shop/publications/order

Related tasks:

- “Printing DB2 books from PDF files” on page 325
- “Finding topics by accessing the DB2 Information Center from a browser” on page 328
- “Viewing technical documentation online directly from the DB2 HTML Documentation CD” on page 331

Related reference:

- “Overview of DB2 Universal Database technical information” on page 317

Accessing online help

The online help that comes with all DB2 components is available in three types:

- Window and notebook help
- Command line help
- SQL statement help

Window and notebook help explain the tasks that you can perform in a window or notebook and describe the controls. This help has two types:

- Help accessible from the **Help** button
- Infopops

The **Help** button gives you access to overview and prerequisite information. The infopops describe the controls in the window or notebook. Window and notebook help are available from DB2 centers and components that have user interfaces.

Command line help includes Command help and Message help. Command help explains the syntax of commands in the command line processor. Message help describes the cause of an error message and describes any action you should take in response to the error.

SQL statement help includes SQL help and SQLSTATE help. DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the syntax of SQL statements (SQL states and class codes).

Note: SQL help is not available for UNIX operating systems.

Procedure:

To access online help:

- For window and notebook help, click **Help** or click that control, then click **F1**. If the **Automatically display infopops** check box on the **General** page of the **Tool Settings** notebook is selected, you can also see the infopop for a particular control by holding the mouse cursor over the control.
- For command line help, open the command line processor and enter:
  - For Command help:
    ```
    ? 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.
- For Message help:
  ```
  ? XXXnnnnn
  ```
  where *XXXnnnnn* represents a valid message identifier.

  For example, `? SQL30081` displays help about the SQL30081 message.
- For SQL statement help, open the command line processor and enter:
  - For SQL help:
    ```
    ? 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.
  - For SQLSTATE help:

```
help statement
```

where *statement* represents an SQL statement.

For example, help SELECT displays help about the SELECT statement.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 328
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 331

Finding topics by accessing the DB2 Information Center from a browser

The DB2 Information Center accessed from a browser enables you to access the information you need to take full advantage of DB2 Universal Database and DB2 Connect. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, metadata, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser is composed of the following major elements:

Navigation tree
The navigation tree is located in the left frame of the browser window. The tree expands and collapses to show and hide topics, the glossary, and the master index in the DB2 Information Center.

Navigation toolbar
The navigation toolbar is located in the top right frame of the browser window. The navigation toolbar contains buttons that enable you to search the DB2 Information Center, hide the navigation tree, and find the currently displayed topic in the navigation tree.

Content frame
The content frame is located in the bottom right frame of the browser window. The content frame displays topics from the DB2 Information Center when you click on a link in the navigation tree, click on a search result, or follow a link from another topic or from the master index.

Prerequisites:

To access the DB2 Information Center from a browser, you must use one of the following browsers:

- Microsoft Explorer, version 5 or later
- Netscape Navigator, version 6.1 or later

Restrictions:

The DB2 Information Center contains only those sets of topics that you chose to install from the *DB2 HTML Documentation CD*. If your Web browser returns a `File not found` error when you try to follow a link to a topic, you must install one or more additional sets of topics *DB2 HTML Documentation CD*.

Procedure:

To find a topic by searching with keywords:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

To find a topic in the navigation tree:

1. In the navigation tree, click the book icon of the category of topics related to your area of interest. A list of subcategories displays underneath the icon.
2. Continue to click the book icons until you find the category containing the topics in which you are interested. Categories that link to topics display the category title as an underscored link when you move the cursor over the category title. The navigation tree identifies topics with a page icon.
3. Click the topic link. The topic displays in the content frame.

To find a topic or term in the master index:

1. In the navigation tree, click the "Index" category. The category expands to display a list of links arranged in alphabetical order in the navigation tree.
2. In the navigation tree, click the link corresponding to the first character of the term relating to the topic in which you are interested. A list of terms with that initial character displays in the content frame. Terms that have multiple index entries are identified by a book icon.
3. Click the book icon corresponding to the term in which you are interested. A list of subterms and topics displays below the term you clicked. Topics are identified by page icons with an underscored title.
4. Click on the title of the topic that meets your needs. The topic displays in the content frame.

Related concepts:

- "Accessibility" on page 337
- "DB2 Information Center for topics" on page 339

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 330
- "Updating the HTML documentation installed on your machine" on page 332
- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 334
- "Searching the DB2 documentation" on page 335

Related reference:

- "Overview of DB2 Universal Database technical information" on page 317

Finding product information by accessing the DB2 Information Center from the administration tools

The DB2 Information Center provides quick access to DB2 product information and is available on all operating systems for which the DB2 administration tools are available.

The DB2 Information Center accessed from the tools provides six types of information.

Tasks Key tasks you can perform using DB2.

Concepts
Key concepts for DB2.

Reference
DB2 reference information, such as keywords, commands, and APIs.

Troubleshooting
Error messages and information to help you with common DB2 problems.

Samples
Links to HTML listings of the sample programs provided with DB2.

Tutorials
Instructional aid designed to help you learn a DB2 feature.

Prerequisites:

Some links in the DB2 Information Center point to Web sites on the Internet. To display the content for these links, you will first have to connect to the Internet.

Procedure:

To find product information by accessing the DB2 Information Center from the tools:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter **db2ic**.
2. Click the tab of the information type related to the information you are attempting to find.
3. Navigate through the tree and click on the topic in which you are interested. The Information Center will then launch a Web browser to display the information.
4. To find information without browsing the lists, click the **Search** icon to the right of the list.

   Once the Information Center has launched a browser to display the information, you can perform a full-text search by clicking the **Search** icon in the navigation toolbar.

Related concepts:

- "Accessibility" on page 337
- "DB2 Information Center for topics" on page 339

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 328
- "Searching the DB2 documentation" on page 335

Viewing technical documentation online directly from the DB2 HTML Documentation CD

All of the HTML topics that you can install from the *DB2 HTML Documentation CD* can also be read directly from the CD. Therefore, you can view the documentation without having to install it.

Restrictions:

Because the following items are installed from the DB2 product CD and not the *DB2 HTML Documentation CD*, you must install the DB2 product to view these items:

- Tools help
- DB2 Quick Tour
- Release notes

Procedure:

1. Insert the *DB2 HTML Documentation* CD. On UNIX operating systems, mount the *DB2 HTML Documentation CD*. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start your HTML browser and open the appropriate file:
   - For Windows operating systems:
     ```
     e:\Program Files\sqllib\doc\htmlcd\%L\index.htm
     ```
     where *e* represents the CD-ROM drive, and %L is the locale of the documentation that you wish to use, for example, **en_US** for English.
   - For UNIX operating systems:
     ```
     /cdrom/Program Files/sqllib/doc/htmlcd/%L/index.htm
     ```
     where */cdrom/* represents where the CD is mounted, and %L is the locale of the documentation that you wish to use, for example, **en_US** for English.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 328
- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 334

Related reference:

- "Overview of DB2 Universal Database technical information" on page 317

Updating the HTML documentation installed on your machine

It is now possible to update the HTML installed from the *DB2 HTML Documentation CD* when updates are made available from IBM. This can be done in one of two ways:

- Using the Information Center (if you have the DB2 administration GUI tools installed).
- By downloading and applying a DB2 HTML documentation FixPak .

Note: This will NOT update the DB2 code; it will only update the HTML documentation installed from the *DB2 HTML Documentation CD*.

Procedure:

To use the Information Center to update your local documentation:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter **db2ic**.
2. Ensure your machine has access to the external Internet; the updater will download the latest documentation FixPak from the IBM server if required.
3. Select **Information Center —> Update Local Documentation** from the menu to start the update.
4. Supply your proxy information (if required) to connect to the external Internet.

The Information Center will download and apply the latest documentation FixPak, if one is available.

To manually download and apply the documentation FixPak :

1. Ensure your machine is connected to the Internet.
2. Open the DB2 support page in your Web browser at: www.ibm.com/software/data/db2/udb/winos2unix/support.
3. Follow the link for version 8 and look for the ″Documentation FixPaks″ link.
4. Determine if the version of your local documentation is out of date by comparing the documentation FixPak level to the documentation level you have installed. This current documentation on your machine is at the following level: **DB2 v8.1 GA**.
5. If there is a more recent version of the documentation available then download the FixPak applicable to your operating system. There is one FixPak for all Windows platforms, and one FixPak for all UNIX platforms.
6. Apply the FixPak:
   - For Windows operating systems: The documentation FixPak is a self extracting zip file. Place the downloaded documentation FixPak in an empty directory, and run it. It will create a **setup** command which you can run to install the documentation FixPak.
   - For UNIX operating systems: The documentation FixPak is a compressed tar.Z file. Uncompress and untar the file. It will create a directory named `delta_install` with a script called **installdocfix**. Run this script to install the documentation FixPak.

Related tasks:

- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 334

Related reference:

- "Overview of DB2 Universal Database technical information" on page 317

Copying files from the DB2 HTML Documentation CD to a Web Server

The entire DB2 information library is delivered to you on the *DB2 HTML Documentation CD*, so you can install the library on a Web server for easier access. Simply copy to your Web server the documentation for the languages that you want.

Procedure:

To copy files from the *DB2 HTML Documentation CD* to a Web server, use the appropriate path:

- For Windows operating systems:

  ```
  E:\Program Files\sqllib\doc\htmlcd\%L\*.*
  ```

  where *E* represents the CD-ROM drive and *%L* represents the language identifier.
- For UNIX operating systems:

  ```
  /cdrom:Program Files/sqllib/doc/htmlcd/%L/*.*
  ```

  where *cdrom* represents the CD-ROM drive and *%L* represents the language identifier.

Related tasks:

- "Searching the DB2 documentation" on page 335

Related reference:

- "Supported DB2 interface languages, locales, and code pages" in the *Quick Beginnings for DB2 Servers*
- "Overview of DB2 Universal Database technical information" on page 317

Troubleshooting DB2 documentation search with Netscape 4.x

Most search problems are related to the Java support provided by web browsers. This task describes possible workarounds.

Procedure:

A common problem with Netscape 4.x involves a missing or misplaced security class. Try the following workaround, especially if you see the following line in the browser Java console:

```
Cannot find class  java/security/InvalidParameterException
```

- On Windows operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied `x:Program Files\sqllib\doc\htmlcd\locale\InvalidParameterException.class` file to the `java\classes\java\security\` directory relative to your Netscape browser installation, where *x* represents the CD-ROM drive letter and *locale* represents the name of the desired locale.

  Note: You may have to create the `java\security\` subdirectory structure.
- On UNIX operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied `/cdrom/Program Files/sqllib/doc/htmlcd/locale/InvalidParameterException.class` file to the `java/classes/java/security/` directory relative to your Netscape browser installation, where *cdrom* represents the mount point of the CD-ROM and *locale* represents the name of the desired locale.

  Note: You may have to create the `java/security/` subdirectory structure.

If your Netscape browser still fails to display the search input window, try the following:

- Stop all instances of Netscape browsers to ensure that there is no Netscape code running on the machine. Then open a new instance of the Netscape browser and try to start the search again.
- Purge the browser's cache.
- Try a different version of Netscape, or a different browser.

Related tasks:

- "Searching the DB2 documentation" on page 335

Searching the DB2 documentation

To search DB2's documentation, you need Netscape 6.1 or higher, or Microsoft's Internet Explorer 5 or higher. Ensure that your browser's Java support is enabled.

A pop-up search window opens when you click the search icon in the navigation toolbar of the Information Center accessed from a browser. If you are using the search for the first time it may take a minute or so to load into the search window.

Restrictions:

The following restrictions apply when you use the documentation search:

- Boolean searches are not supported. The boolean search qualifiers *and* and *or* will be ignored in a search. For example, the following searches would produce the same results:
  - servlets *and* beans
  - servlets *or* beans
- Wildcard searches are not supported. A search on *java** will only look for the literal string *java** and would not, for example, find *javadoc*.

In general, you will get better search results if you search for phrases instead of single words.

Procedure:

To search the DB2 documentation:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

Note: When you perform a search, the first result is automatically loaded into your browser frame. To view the contents of other search results, click on the result in results lists.

Related tasks:

- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 334

Online DB2 troubleshooting information

With the release of DB2® UDB Version 8, there will no longer be a *Troubleshooting Guide*. The troubleshooting information once contained in this guide has been integrated into the DB2 publications. By doing this, we are able to deliver the most up-to-date information possible. To find information on the troubleshooting utilities and functions of DB2, access the DB2 Information Center from any of the tools.

Refer to the DB2 Online Support site if you are experiencing problems and want help finding possible causes and solutions. The support site contains a large, constantly updated database of DB2 publications, TechNotes, APAR (product problem) records, FixPaks, and other resources. You can use the support site to search through this knowledge base and find possible solutions to your problems.

Access the Online Support site at www.ibm.com/software/data/db2/udb/winos2unix/support, or by clicking the **Online Support** button in the DB2 Information Center. Frequently changing information, such as the listing of internal DB2 error codes, is now also available from this site.

Related concepts:

- "DB2 Information Center for topics" on page 339

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 330

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features in DB2® Universal Database Version 8:

- DB2 allows you to operate all features using the keyboard instead of the mouse. See "Keyboard Input and Navigation".
- DB2 enables you customize the size and color of your fonts. See "Accessible Display" on page 338.
- DB2 allows you to receive either visual or audio alert cues. See "Alternative Alert Cues" on page 338.
- DB2 supports accessibility applications that use the Java™ Accessibility API. See "Compatibility with Assistive Technologies" on page 338.
- DB2 comes with documentation that is provided in an accessible format. See "Accessible Documentation" on page 338.

Keyboard Input and Navigation

Keyboard Input

You can operate the DB2 Tools using only the keyboard. You can use keys or key combinations to perform most operations that can also be done using a mouse.

Keyboard Focus

In UNIX-based systems, the position of the keyboard focus is highlighted, indicating which area of the window is active and where your keystrokes will have an effect.

Accessible Display

The DB2 Tools have features that enhance the user interface and improve accessibility for users with low vision. These accessibility enhancements include support for customizable font properties.

Font Settings

The DB2 Tools allow you to select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

Non-dependence on Color

You do not need to distinguish between colors in order to use any of the functions in this product.

Alternative Alert Cues

You can specify whether you want to receive alerts through audio or visual cues, using the Tools Settings notebook.

Compatibility with Assistive Technologies

The DB2 Tools interface supports the Java Accessibility API enabling use by screen readers and other assistive technologies used by people with disabilities.

Accessible Documentation

Documentation for the DB2 family of products is available in HTML format. This allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

DB2 tutorials

The DB2® tutorials help you learn about various aspects of DB2 Universal Database. The tutorials provide lessons with step-by-step instructions in the areas of developing applications, tuning SQL query performance, working with data warehouses, managing metadata, and developing Web services using DB2.

Before you begin:

Before you can access these tutorials using the links below, you must install the tutorials from the *DB2 HTML Documentation* CD-ROM.

If you do not want to install the tutorials, you can view the HTML versions of the tutorials directly from the *DB2 HTML Documentation CD*. PDF versions of these tutorials are also available on the *DB2 PDF Documentation CD*.

Some tutorial lessons use sample data or code. See each individual tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

If you installed the tutorials from the *DB2 HTML Documentation* CD-ROM, you can click on a tutorial title in the following list to view that tutorial.

Business Intelligence Tutorial: Introduction to the Data Warehouse Center
Perform introductory data warehousing tasks using the Data Warehouse Center.

Business Intelligence Tutorial: Extended Lessons in Data Warehousing
Perform advanced data warehousing tasks using the Data Warehouse Center.

Development Center Tutorial for Video Online using Microsoft® Visual Basic
Build various components of an application using the Development Center Add-in for Microsoft Visual Basic.

Information Catalog Center Tutorial
Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Video Central for e-business Tutorial
Develop and deploy an advanced DB2 Web Services application using WebSphere® products.

Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 Information Center for topics

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 Universal Database™ and DB2 Connect™ in your business. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, the Information Catalog Center, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser has the following features:

Regularly updated documentation
Keep your topics up-to-date by downloading updated HTML.

Search
Search all of the topics installed on your workstation by clicking **Search** in the navigation toolbar.

Integrated navigation tree
Locate any topic in the DB2 library from a single navigation tree. The navigation tree is organized by information type as follows:

- Tasks provide step-by-step instructions on how to complete a goal.
- Concepts provide an overview of a subject.
- Reference topics provide detailed information about a subject, including statement and command syntax, message help, requirements.

Master index
Access the information in topics and tools help from one master index. The index is organized in alphabetical order by index term.

Master glossary
The master glossary defines terms used in the DB2 Information Center. The glossary is organized in alphabetical order by glossary term.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 328
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 330
- "Updating the HTML documentation installed on your machine" on page 332

Appendix C. 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/region 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/region 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

Trademarks

Index

Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-237-5511 for customer service
- 1-888-426-4343 to learn about available service options
- 1-800-IBM-4YOU (426-4968) for DB2 marketing and sales

In Canada, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-800-465-9600 to learn about available service options
- 1-800-IBM-4YOU (1-800-426-4968) for DB2 marketing and sales

To locate an IBM office in your country or region, check IBM's Directory of Worldwide Contacts on the web at www.ibm.com/planetwide

Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at www.ibm.com/software/data/db2/udb

This site contains the latest information on the technical library, ordering books, client downloads, newsgroups, FixPaks, news, and links to web resources.

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) to order products or to obtain general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, go to the IBM Worldwide page at www.ibm.com/planetwide

Part Number: CT17UNA

Printed in U.S.A.

SC09-4827-00

(1P) P/N: CT17UNA

Spine information:

IBM® DB2 Universal Database™ Programming Server Applications Version 8