IBM® DB2 Universal Database™

Application Development Guide: Programming Server Applications

Version 8.2

SC09-4827-01

IBM® DB2 Universal Database™

Application Development Guide: Programming Server Applications

Version 8.2

SC09-4827-01

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.
  - The DB2 .NET Data Provider, and the OLE DB .NET and ODBC .NET data providers.
  - Java (both JDBC and SQLJ) and considerations for building Java applications for use on 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 about programming using server-side objects, 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.
    - Creating routines, including external routines, and the CREATE statement.
    - Procedure parameter modes and parameter handling.
    - Procedure result sets.
    - SQL procedures including debugging and condition handling.
    - User-defined scalar and table functions.
    - User-defined scalar and table function calls (FIRST call, FINAL call,...) and scratchpads.
    - Methods.
    - Authorizations and binding of external routines.
    - Language-specific considerations for C, Java, .NET common language runtime, and OLE automation routines.
    - Invoking routines.
    - Function selection.
    - 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:

– 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 set up your application development environment, including specific instructions for Java and WebSphere MQ functions.
– How to set up the sample database
– How to migrate your applications from previous versions of DB2.
– 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 C# and Visual Basic .NET appllcations and CLR .NET routines on 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

Chapter 1. Introduction to routines

Routines in application development

A routine is a database object that can encapsulates programming and database logic related to a specific task. There are three types of routines: procedures, functions, and methods. Each type of routine provides a different interface for containing logic and database operations that can be used to extend the functionality of an SQL statement or a client application. You should consider the many benefits of creating and using routines when you are developing or updating a database application.

When faced with the task of developing new functionality that will 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.

Benefits of using routines:

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

Encapsulate application logic
In an environment with numerous client computers, each running a variety of database applications, the effective use of routines can simplify code reuse, code standardization, and code 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 that encapsulates that behavior, will require modification. If routines had not been used in this instance, application logic changes would have been required in each client application.

Enable controlled access to database objects
You can use routines to control access to database objects. A user might not have permission to generally issue a particular SQL statement, however the user can be given permission to invoke routines that contain specific implementations of these statements.

Reduce network traffic
When an application is running on a client computer, each SQL statement is sent separately from the client computer to the server computer and each result is returned separately. This can result in a high degree of network traffic. If a piece of work can be identified that involves heavy database activity and little user interaction, it makes sense to install this piece of work on the server. With this work running on the server, the quantity of network traffic between the client computer and the server computer is reduced. DB2 routines run on the database server in this manner. Using routines is an effective way of reducing network traffic and improving overall client application performance.

Alleviate the processing load on the client
In environments where the performance of a client computer is a concern, routines are a practical means of reducing the dependence on the client computer. After an application invokes a routine, the processing of the routine is done on the database server, thus allowing the application to exploit the power of the database server while relieving the client computer of the processing load.

Allow faster, more efficient execution
Routines are database objects and therefore have a closer relationship with the database manager than client applications do. For some types of routines the performance of SQL statements can be much better than the performance of SQL statements that are executed from a client application. For example, NOT FENCED routines run in the same process as the database manager using shared memory for communication. This makes the routines more proficient in transmitting SQL requests and data, than a client application could ever be that communicates using TCP/IP protocols.

Interoperability of logic implementations
Because code modules are often implemented by different programmers, each with programming expertise in different programming languages, and because it is generally desirable to reuse code wherever possible to save on development time and costs, DB2® routines are highly interoperable.

- A client application in one programming language can invoke routines that are implemented in a different programming language. For example, C client applications can invoke .NET common language runtime routines.
- A routine can invoke another routine regardless of the routine type or the implementation language of the routine. For example a Java™ procedure (one type of routine) can invoke an SQL scalar function (another type of routine with a different implementation language).
- A routine created in a database server on one operating system can be invoked from a DB2 client running on a different operating system.

There are various kinds of routines that address particular functional needs and various routine implementations. The choice of routine type and implementation can impact the degree to which the above benefits are exhibited. In general, routines are a powerful way of encapsulating logic so that you can extend your SQL, and improve the structure, maintenance, and potentially the performance of your applications.

Related concepts:
- “Procedures” on page 11
- “Routine invocation” on page 193
- “Supported routine programming languages” on page 19
- “User-defined scalar functions” on page 13
- “Methods” on page 16
- “User-defined scalar functions” on page 15

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 UNIX C routines” in the *Application Development Guide: Building and Running Applications*
- “Building UNIX C++ routines” 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 UNIX Micro Focus COBOL routines” 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*
- “Writing routines” on page 33
- “Creating routines in the database” on page 31
- “Debugging routines” on page 38

Types of routines (procedures, functions, methods)

Routines are grouped primarily by their functionality however they can also be grouped by their implementation. There are three main functional types of routines: procedures (also called stored procedures), functions, and methods. There are a few possible implementations for routines including: built-in, sourced, SQL, and external. The functional types of routines are discussed first and explanations of the possible implementations follows.

Functional types of routines:

Procedures

Procedures, also called stored procedures, serve as sub-routine extensions to client applications, routines, triggers, and dynamic compound statements. Procedures are invoked by executing the CALL statement with a reference to a procedure.

Functions

A function is a relationship between a set of input data values and a set of result values. Functions enable you to extend and customize SQL. Functions are invoked from within elements of SQL statements such as a select-list or a FROM clause. There are four types of functions: aggregate functions, scalar functions, row functions, and table functions.

Aggregate functions

Also called a column function, this type of function returns a scalar value that is the result of an evaluation over a set of like input values. The similar input values can, for example, be specified by a column within a table, or by tuples in a VALUES clause. This set of values is called the argument set. For example, the following query finds the total quantity of bolts that are in stock or on order including all kinds of bolts by using the aggregate function SUM:

```
SELECT SUM(qinstock + qonorder)
FROM inventory
WHERE description LIKE '%Bolt%'
```

An aggregate function cannot be implemented as an external function; only as a sourced function that has been sourced off of a built-in aggregate function.

Scalar functions
A scalar function is a function that, for each set of one or more scalar parameters, returns a single scalar value. Examples of scalar functions include `length`, and `substr`. A scalar function can also be created that does a complex mathematical calculation on the input parameters. Scalar functions can be referenced anywhere that an expression is valid within an SQL statement, such as in a select-list, or in a FROM clause. Scalar functions can be implemented as either external or sourced functions.

Row functions
A row function is a function, that for each set of one or more scalar parameters, returns a single row. Row functions can only be used as a transform function mapping attributes of a structured type into built-in data type values in a row. A row function can only be implemented as an SQL function.

Table functions

Table functions are functions, that for a group of sets of one or more parameters, returns a table to the SQL statement that references it. Table functions can only be referenced in the FROM clause of a SELECT statement. The table that is returned by a table function can participate in joins, grouping operations, set operation such as UNION, and any operation that could be applied to a read-only view. Table functions can be implemented in SQL, or in an external programming language.

Methods
An encapsulation of logic that provides behavior for 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 method is generally implemented for a structured type to represent an operation on the attributes of the structured type. For a geometric shape a method might calculate the volume of the shape. Methods can be implemented as an SQL or external method.

The following diagram illustrates the classification hierarchy of routines:

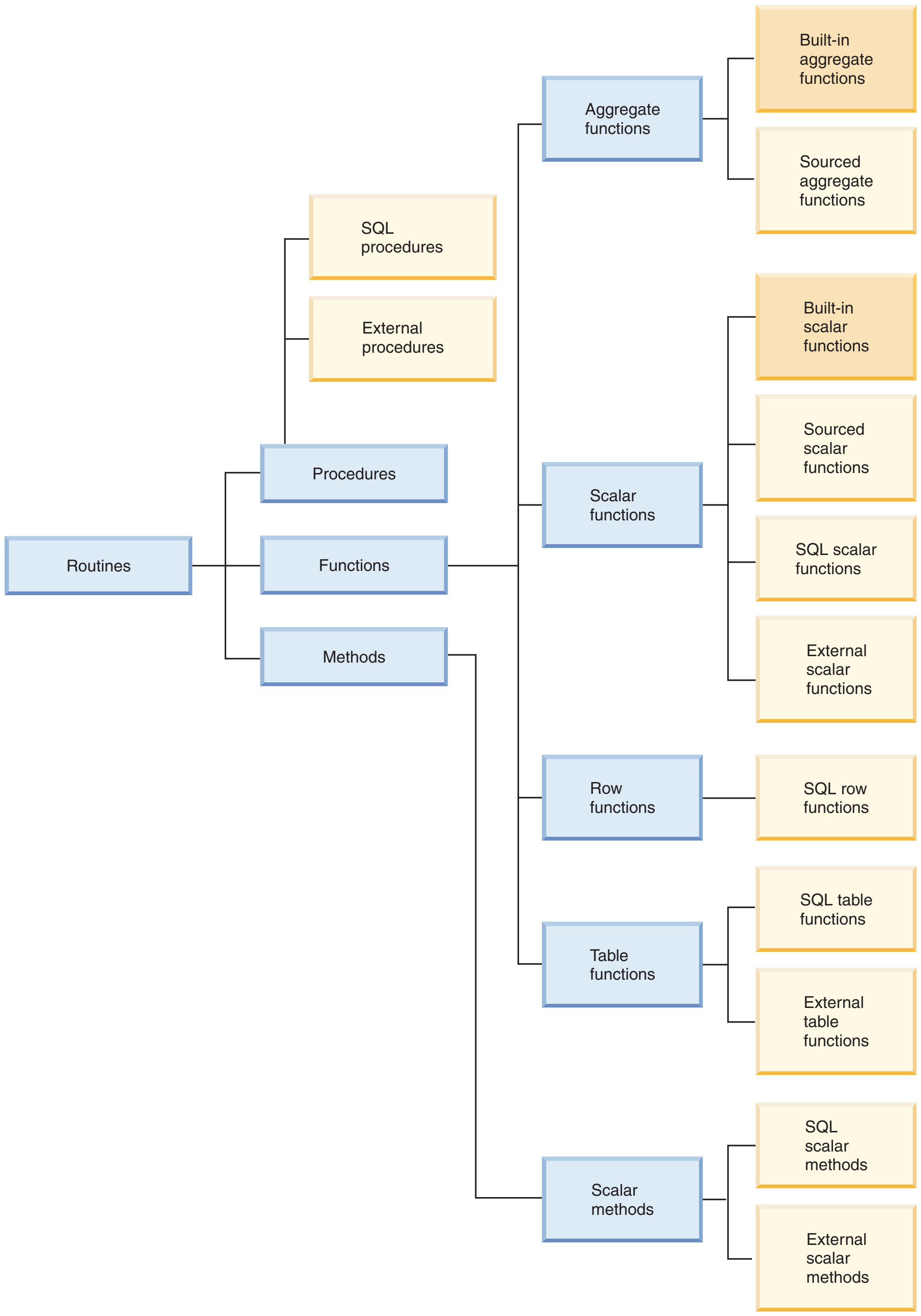


Figure 1. Classifications of routines

Types of routine implementations:

There are a few possible implementations for routines: built-in, sourced, SQL, and external.

Built-in

Some routines are built into the code of the DB2 system. These routines are strongly typed and perform well because their logic is native to the database code. These routines are found in the SYSIBM schema. Some examples of built-in scalar and aggregate functions include:

Built-in scalar functions
+, −, *, /, ‖, substr, concat, length, char, decimal days

Built-in aggregate functions
avg, count, min, max, stdev, sum, variance

Built-in functions comprise most of the commonly required casting, string manipulation, and arithmetic functionality. You can immediately use these functions in your SQL statements. For a complete list of available built-in functions, see the SQL Reference.

The other implementations are user motivated implementations. These implementations, unlike built-in functions, require that the user explicitly create the routine using the appropriate CREATE statement for the routine type. User created functions and procedures are located in the SYSTOOLS schema.

Sourced
A sourced function is a function that duplicates the semantics of another function, called its source function. Currently only scalar and aggregate functions can be sourced functions. Sourced functions are particularly useful for allowing a distinct type to selectively inherit the semantics of its source type. Essentially, sourced functions are a special form of an SQL implementation for a function.

SQL

An SQL routine is composed entirely of SQL statements. You specify these statements in the CREATE statement that you use to create the routine in the database. SQL Procedural Language (SQL PL) is a language extension of basic SQL that consists of statements and language elements that can be used to implement programming logic in SQL.

SQL PL includes a set of statements for declaring varables and condition handlers (DECLARE statement), assigning values to variables (assignment-statement), and for implementing procedural logic (control-statements: IF, WHILE, FOR, GOTO, LOOP, SIGNAL, and others). SQL including SQL PL, or where restricted a subset of SQL PL, can be used to create SQL procedures, functions, and methods.

External
A routine that is created in the database using the routine type-specific CREATE statement, but that has its routine logic implemented in an external host programming language application. The association of the routine with the external code application is asserted by the specification of the EXTERNAL clause in the CREATE statement. External routines can be written in C, C++, Java™, OLE, and .NET common language runtime supported programming languages. External procedures can additionally be written in COBOL.

DB2 provided routines:

DB2 additionally provides some procedures and functions in the SYSPROC, SYSFUN, and SYSTOOLS schemas that you can use. Although these additional routines are shipped with DB2, they are not built-in routines. Instead they are implemented as pre-installed user-defined routines. These routines typically encapsulate a utility function. Some examples of these include: `SNAPSHOT_TABLE`, `HEALTH_DB_HI`, `SNAPSHOT_FILEW`, `REBIND_ROUTINE_PACKAGE`. You can immediately use these functions and procedures provided that you have the SYSPROC schema and SYSFUN schema in your CURRENT PATH. These schemas are included in the CURRENT PATH by default.

Related concepts:

- "Routines in application development" on page 3
- "Performance considerations for developing routines" on page 22
- "Security considerations for routines" on page 24

User-defined routines

DB2® provides built-in routines that capture the functionality of most commonly used arithmetic, string, and casting functions, however to encapsulate logic of your own, DB2 allows you to create your own routines. These routines are said to be user-defined. You can create your own procedures, functions and methods in any of the supported implementation styles for the routine type. Generally the prefix 'user-defined' is not used when referring to procedures and methods, however user-defined functions are commonly called UDFs.

Routine CREATE statements:

User-defined procedures, functions and methods are created in the database by executing the appropriate CREATE statement for the routine class. These routine creation statements include: CREATE PROCEDURE, CREATE FUNCTION, and CREATE METHOD. The clauses specific to each of the CREATE statements define characteristics of the routine, such as the routine name, the number and type of routine arguments, and details about the routine logic. DB2 uses the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in DB2's system tables that users can query. Executing the CREATE statement to create a routine is also referred to as defining a routine or registering a routine.

Routine logic implementation:

There are a three implementation styles that can be used to specify the logic of a routine: SQL, external, and sourced. These implementations are compared below so that you can see the benefits and utility of each implementation:

SQL

An SQL routine's logic is written entirely in SQL that is specified within the body of the CREATE statement of the routine. An SQL-procedure-body, SQL-function body, or SQL-method body can be composed of SQL and SQL PL statements.

SQL routines are quick and easy to implement because of their simple syntax and perform well due to their close relationship with DB2 for implementations that contain mostly SQL statements and less complex uses of SQL PL logic. In the case of SQL procedures they also provide easy to use error handling support. SQL routine are however limited in that they cannot directly effect system calls, cannot perform operations on entities that reside outside of the database, and depending of the functional type of the routine may not support the execution of all SQL statements.

External

External routines are routines that have their logic implemented in a user-created library or class that resides on the filesystem of the database server - external to the database itself. The library or class can be compiled from a source application written in one of the following host programming languages: C, C++, Java™, OLE, or any .NET compatible language. External procedures can also be written in COBOL. All external routines can contain SQL, except OLE routines.

External routines are generally a bit more complex to implement than SQL routines, however they are extremely powerful because with an external routine you can harness the full functionality and performance of the chosen implementation programming language. External functions also have the advantage of being able to access and manipulate entities that reside outside of the database such as a network or filesystem. For routines that require a smaller degree of interaction with the DB2 database, but that must contain a lot of logic or more complex logic, an external routine implementation is preferable. As an example, external routines are ideal to use to implement new functions that operate on and enhance the utility of built-in data types, such as a new string function that operates on a VARCHAR datatype or a complicated mathematical function that operates on a DOUBLE datatype. They are also ideal for logic that might involve an external action, such as sending an email. If you are already comfortable programming in one of the supported programming languages and need to encapsulate logic with a greater emphasis on programming logic than on database accesses, then once you learn the simple steps involved in creating these routines, you will soon discover just how powerful they can be.

Sourced

The sourced implementation style is only applicable to functions. The function logic of a sourced function is derived from an existing source function. The source function is identified by simply specifying the SOURCE clause in the special `CREATE FUNCTION (Sourced or template)` statement. There is no language particularly associated with this implementation. The most common use of sourced functions is to allow a distinct type to selectively inherit some of the functions and operators that apply to its source type. Sourced functions are simple to implement and are useful if you want to rename an existing function.

Overview of the development of user-defined routines:

For assistance in developing routines, you can 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. Alternatively, you can also develop user-defined routines through the DB2 command line processor.

The development of user-defined routines involves the following tasks:

1. Create the routine in the database. This task, also known as defining or registering a routine, can occur at any time before you invoke the routine, except in the following circumstances:
   - For Java routines that reference an external JAR file or files, the external code and JAR files must be coded and compiled before the routine is created in the database using the routine type specific CREATE statement.

- Routines that execute SQL statements and refer to themselves directly must be created in the database by issuing the CREATE statement before the external code associated with the routine is 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. For external routines, code the routine logic. The logic of SQL routines is contained within the CREATE statement of the SQL routine.
3. For external routines, build (precompile -- for routines with embedded SQL, compile, and link) the routine. (See the related links for operating system and language-specific build information.)
4. Debug and test the routine.
5. Grant the EXECUTE privilege on the routine to the routine invoker or invokers
6. Invoke the routine.

Related concepts:
- “Procedures” on page 11
- “SQL in external routines” on page 101
- “Types of routines (procedures, functions, methods)” on page 5
- “User-defined table functions” on page 57
- “User-defined scalar functions” on page 13
- “Methods” on page 16
- “SQL access levels in SQL routines” on page 63
- “SQL Procedural Language (SQL PL) in DB2” on page 61

Comparison of procedures, functions, and methods

There are three types of routines you can develop: procedures, user-defined functions (UDFs), and methods. While the details involved in creating and implementing them are similar, they each serve different purposes.

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

Procedures

A procedure, also called a stored procedure, is a database object created by executing the CREATE PROCEDURE statement that can encapsulates logic and SQL statements. Procedures are used as subroutine extensions to applications, and other database objects that can contain logic.

Features

- Enables the encapsulation of SQL statements, function invocations, and logic elements that formulate a particular subroutine module that can be reused.
- Procedures can be called from client applications, other routines, triggers and dynamic compound statements. Procedures are called using the CALL statement.
- Procedures can return multiple result sets.
- Procedures can contain SQL statements that read or modify table data in both single and multiple partition databases.

- When a procedure is invoked the SQL and logic within a procedure is executed on the server. Data is only transferred between the client and the database server in the procedure call and in the procedure return. If you have a series of SQL statements to execute within a client application, and the application does not need to do any processing in between the statements, then this series of statements would benefit from being included in a procedure.

  Note: If only one SQL statement is invoked in a procedure, the overhead of setting up this invocation outweighs the benefit in network traffic savings.

Limitations

- Procedures are not intended to be called from within elements of an SQL query. Procedures can only be invoked by using the CALL statement where it is supported. Functions can be used to express logic that transforms column values. Although procedures can return result sets, table functions can be used to return a table within the FROM clause of an SQL query.
- Output arguments of procedure calls cannot be directly used by another SQL statement.
- Procedures cannot preserve state between invocations.

Common uses

- To implement application sub-routines that specifically encapsulate the database logic associated with a particular task. For example, a business application for managing employee information could use a procedure to encapsulate the database operations involved in hiring an employee.

  Such a procedure could insert employee information into an employee table, a department table, and a benefits table, calculate the employee's weekly pay amount based on an input parameter, and return the weekly pay value as one of the output parameters. Another procedure could contain a statistical analysis of data in the employee table and return result sets that contain the results of the analysis. This use of procedures effectively isolates database tasks from non-database tasks within an application.
- Standardize application logic. If multiple applications must similarly access or modify the database, a procedure can provide a single interface for that access or modification. The procedure is available to be used by all of the applications. If the interface must change to accommodate a change in business logic, only the single procedure must be modified.

Supported languages

- SQL
- C/C++
- Java™
- OLE
- COBOL
- .NET common language runtime languages

Note: SQL procedures are supported natively and do not require the installation of a compiler.

Related concepts:

- “Routines in application development” on page 3
- “Procedure parameter modes” on page 42
- “Procedure result sets” on page 42
- “SQL Procedural Language (SQL PL) in DB2” on page 61

Related tasks:
- “Setting up the application development environment” in the *Application Development Guide: Building and Running Applications*
- “Creating SQL procedures from the command line” on page 66
- “Calling procedures from triggers or SQL routines” on page 202
- “Calling procedures from applications or external routines” on page 200

Related reference:
- “CALL statement” in the *SQL Reference, Volume 2*
- “CREATE PROCEDURE statement” in the *SQL Reference, Volume 2*
- “DB2 supported development software” in the *Application Development Guide: Building and Running Applications*

User-defined scalar functions

Scalar user-defined functions (UDFs) enable you to extend and customize SQL statements. They can be invoked in the same manner as DB2® built-in functions such as LENGTH and COUNT. 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 upon each invocation.

SQL scalar user-defined functions:

SQL scalar UDFs enable you to encapsulate SQL statements, built-in functions and other routine references, and a subset of SQL PL statements that can be used to implement some basic database logic. SQL scalar functions can read and modify SQL data. SQL functions give the best performance when they make use of built-in functions and do not contain extremely complex logic. For extremely complex logic, consider implementing an external scalar UDF.

External scalar user-defined functions:

External scalar UDFs have their logic implemented in an external programming language. The logic of the function can access the filesystem, perform system calls or access a network. The execution of the external scalar UDF routine logic, like that of SQL scalar UDFs takes place on the server. External scalar UDFs can read SQL data, but cannot modify SQL data. An external scalar UDF can be repeatedly invoked for a single reference of the function and can maintain state between these invocations by using a scratchpad, which is a memory buffer. This can be powerful if a function requires some initial, but expensive, setup logic. The setup logic can be done on a first invocation that may make use of the scratchpad to store some values that can be accessed or updated in subsequent invocations of the scalar function.

Features of SQL and external scalar UDFs

- Can be referenced as part of an SQL statement anywhere an expression is supported.

- The output of a scalar UDF can be used directly by the invoking SQL statement.
- For external scalar user-defined functions, state can be maintained between the iterative invocations of the function by using a scratchpad.
- Can provide a performance advantage when used in predicates, because they are executed at the server. If a function can be applied to a candidate row at the server, it can often eliminate the row from consideration before transmitting it to the client machine, reducing the amount of data that must be passed from server to client.
- An excellent way to build scalar functions out of existing built-in functions. For example, you can create a complex mathematical formula by re-using the built-in scalar functions along with other logic.

Limitations

- Cannot do transaction management within a scalar UDF. That is, you cannot issue a COMMIT or a ROLLBACK within a scalar UDF.
- Cannot return result sets.
- Scalar UDF's are intended to return a single scalar value per set of inputs.
- External scalar UDF's are not intended to be used for a single invocation. They are designed such that for a single reference to the UDF and a given set of inputs, that the UDF be invoked once per input, and return a single scalar value. On the first invocation, scalar UDFs can be designed to do some setup work, or store some information that can be accessed in subsequent invocations. SQl scalar UDFs are better suited to functionality that requires a single invocation.
- In a single partition database external scalar UDFs can contain SQL statements. These statements can read data from tables, but cannot modify data in tables. If the database has more than one partition then there must be no SQL statements in an external scalar UDF.

  In serial and in partitioned databases SQL scalar UDFs can contain SQL statements that read data from database tables

Common uses

- Extend the set of DB2 built-in functions.
- Perform logic inside an SQL statement that SQL cannot natively perform.
- Encapsulate a scalar query that is commonly reused as a subquery in SQL statements. 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
- .NET common language runtime languages

Notes:

1. There is a limited capability for creating aggregate functions. Also known as column functions, these functions receive a set of like values (a column of data) and return a single answer. A user-defined aggregate function can only be created if it is sourced upon a built-in aggregate function. For example, if a

distinct type SHOESIZE exists that is defined with base type INTEGER, you could define a UDF, AVG(SHOESIZE), as an aggregate function sourced on the existing built-in aggregate 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 in application development" on page 3
- "Scratchpads for UDFs and methods" on page 52

Related tasks:

- "Invoking scalar functions or methods" on page 211

Related reference:

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

User-defined scalar functions

Like scalar UDFs, a table UDF enables you to extend and customize SQL, but for the purpose of generating a table. 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.

Table functions are powerful because they enable you to make almost any source of data appear as a DB2® table. A table function can be easily created by writing a program that collects the desired data, filters it according to some input parameters if so desired, and returns it to the DB2 one row at a time.

Features

- Can be referenced as part of an SQL statement FROM clause.
- External table-functions can make operating system calls, read data from files or even access data across a network in a single partitioned database.
- Results can be directly processed by the SQL statement that references the table function.
- SQL table functions can encapsulate SQL statements that modify SQL table data. (Only SQL table functions have this property)
- For a single table function reference, a table function can be invoked multiple times and maintain state between invocations by using a scratchpad.
- Provides a set of data for processing.

Limitations

- Cannot do transaction management. This means that you cannot execute COMMIT or ROLLBACK statements from within a table function.
- Cannot return result sets.
- Not designed for single invocations.
- Can only be used in a FROM clause.
- External table functions can read SQL data, but cannot modify SQL data. SQL table functions can be used to contain statements that modify SQL data.

Common uses

- Encapsulate a complex, but commonly used subquery.
- 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.

Supported languages

- SQL
- C/C++
- Java™
- OLE
- OLE DB
- .NET common language runtime languages

Related concepts:

- "Routines in application development" on page 3
- "Scratchpads for UDFs and methods" on page 52
- "Table function processing model" on page 57

Related reference:

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

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:

Features

- 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 in application development" on page 3
- "Scratchpads for UDFs and methods" on page 52

Related tasks:

- “Defining behavior for structured types” on page 251

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 NOT FENCED C/C++ routines.
- SQL routines are written completely in SQL, and can include SQL Procedural Language (SQL PL) elements which is a high level, easy to use languge which makes them quick to implement.
- SQL routines are considered 'safe' by DB2® as they consist entirely of SQL statements. 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 disrupt the functionning of DB2's database engine by causing 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 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, Java Native Interface (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 might be in use on the system if the Java heap of a particular db2fmp process is approaching exhaustion).
- NOT FENCED Java routines are currently not supported. A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.

.NET common language runtime languages

- .NET common language runtime (CLR) routines are routines that are compiled into intermediate language (IL) byte code that can be interpreted by the CLR of the .NET Framework. The source code for a CLR routine can be written in any .NET Framework supported language.
- Working with .NET CLR routines allows the user the flexibility to code in the .NET CLR supported programming language of their choice.
- CLR assemblies can be built up from sub-assemblies that were compiled from different .NET programming language source code, which allows the user to re-use and integrate code modules written in various languages.
- CLR routines can only be created as FENCED NOT THREADSAFE routines. This minimizes the possibility of engine corruption, but also means that these routines cannot benefit from the performance opportunity that can be had with NOT FENCED routines.

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 define 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 might 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 DB automated table functions do not scale well.

Related concepts:

- "Performance considerations for developing routines" on page 22
- "Security considerations for routines" on page 24
- "C/C++ routines" on page 151
- "Java routines" on page 167
- "SQL Procedural Language (SQL PL) in DB2" on page 61

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 UNIX C routines" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX C++ routines" 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*

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 address space.

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 routines″.) If the routine is not safe enough to run in the database manager's process, use the FENCED clause when registering the routine. To limit the creation and running of potentially unsafe code, DB2 requires that a user have a special privilege, CREATE_NOT_FENCED_ROUTINE in order to create NOT FENCED routines.

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 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 can adversely affect other routine threads in the process, causing them to trap, hang, or have damaged data. After one thread abends, the 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 cannot 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

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 can 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.

| NOT FENCED Java routines are currently not supported. A Java routine
| defined as NOT FENCED will be invoked as if it had been defined as
| FENCED THREADSAFE.

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 routines″), and thoroughly testing your routine.

SQL routines

SQL routines, particularly SQL procedures, 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. UDFs that contain complex logic will generaly run more quickly if written in C than in SQL. If the logic is simple, than an SQL UDF will be comparable to any external UDF.

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. The logic of other UDFs and methods might also require that intermediate values 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 in the routine definition. 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″ to a routine that expects a CHAR(200) parameter, 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 same string, ″A SHORT STRING,″ to a routine that expects a VARCHAR(200) parameter results in DB2 simply passing the 14 character s 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 24
- “C/C++ routines” on page 151
- “Java routines” on page 167
- “Restrictions on using routines” on page 29
- “Library and class management considerations” on page 27
- “Improving the performance of SQL procedures” on page 75

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. The greatest potential for corruption arises when code does not properly manage memory or incorrectly uses static variables. These problems are prevalent in languages other than Java™ and .NET programming langauges.

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.

Note: NOT FENCED routines are not supported in Common Criteria compliant configurations.

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 read or 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 in application development" on page 3
- "Performance considerations for developing routines" on page 22
- "Restrictions on using routines" on page 29
- "Library and class management considerations" on page 27

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

When developing routines for DB2®, you have the option of using a variety of different programming languages, including SQL, Java™, C, C++, and .NET compatible languages. If you develop routines in a language other than SQL, they are known as external routines. The compiled source code for an external routine is referred to as a routine body.

Protecting routine bodies

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 statement used to create a routine in the database adds routine definition information to the database catalogs including information about where the external code librarary associated with the routine resides. This is specified in the EXTERNAL clause in the CREATE statement. The routine library or class specified in the EXTERNAL clause is not stored in the database, but resides in the file system of the server. It is imperative for the successful invocation of your external routines that the library associated with a given routine exist in the location specified in the EXTERNAL clause. It is possible that the library can be moved or deleted. If this happens the routine can no longer be invoked successfully.

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.

Note: The bodies of SQL routines are considered to be 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.

The scope of routine bodies

For routines to be used in a database, they must be cataloged with that same database. If there are multiple databases in an instance, you can catalog external routines in one database using routine bodies that are already being used in another database. Hence, the scope of routine bodies is instance wide. While this affords the possibility of reusing code, library or class name conflicts can arise in situations where code is not being reused.

Specifically, library or class name conflicts can manifest themselves in a situation such as the following: there are multiple databases in a single instance and the routines in each database use their own libraries and classes of routine bodies. A conflict arises when the name of a library or class used by a routine in one database is identical to the name of a library or class used by a routine in another database (in the same instance). This is because routine bodies are normally stored in the sqllib/function directory, which is used by all the databases of an instance.

For non-Java routines library name conflicts can be resolved with the following steps:

1. Store the libraries with routine bodies in separate directories for each database.
2. Catalog the routines with the EXTERNAL NAME clause, specifying the full path of the given library.

For Java routines class name conflicts are not solved by moving the files in question into different directories, because the CLASSPATH environment variable is instance-wide. The first class encountered in the CLASSPATH is the one that is used. Therefore, if you have two different Java routines that reference a class with the same name, one of the routines will use the incorrect class. There are two possible solutions: either rename the affected classes, or create a separate instance for each database.

Updating a routine body

If you need to change the body of a routine, do not recompile and relink the routine to the same file (for example, sqllib/function/foo.a) the current routine is using while the database manager is running. If a current routine invocation is accessing a cached version of the routine process and the underlying library is replaced, this can cause the rotine invocation 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. Bind the new routine body (if it contains embedded SQL) with the database.
3. 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.

Note: If the routine body to be updated is used by routines cataloged in multiple databases, the actions prescribed in this section must be completed for each affected database.

Library management-related performance considerations

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 whether they are invoked in the same time frame. Consider a scenario where in a number of applications a call to the procedure ProcA is followed by a call to the procedure ProcB. In this situation, it might 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 a few large libraries.
- The load cost for a library in the C process is paid only once for libraries that are consistently in use by C routines. After the routine's first

invocation, all subsequent invocations, from the same thread in the process, do not need to load the routine’s library.

Routine bodies in partitioned databases

When using external routines in partitioned databases, the library or class must be available on all partitions of the database.

On UNIX®, sqllib/function is a good location for routine bodies, because the sqllib directory is cross-mounted between all partitions of the database.

On Windows®, a good approach would be to create a shared directory accessible to all the partitions, and put the libraries or classes in this directory.

Related concepts:

- “Performance considerations for developing routines” on page 22
- “Security considerations for routines” on page 24
- “Restrictions on using 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 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 on using 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/.fenced` file can create it or write to it. For NOT FENCED routines, the instance owner must have create, read, 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 can also interfere with operation of the JVM for Java routines.
  2. System calls that terminate a process can abnormally terminate one of DB2's processes and result in system or application failure.

     Other system calls can 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 can 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 22
- "Security considerations for routines" on page 24
- "SQL data type handling in C/C++ routines" on page 158

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 190
- "Supported SQL data types in OLE automation" on page 182

Creating routines in the database

| A routine is created in the database when the CREATE statement for that routine is
| executed. For external routines, the CREATE statement execution defines not only
| the name and properties of the routine, but also with the inclusion of the
| EXTERNAL clause points to the location of the external language library that

contains the routine logic. A routine can not be invoked until it has been created in the database. An external routine can not successfully be invoked until it has been created in the database and the library associated with the routine has been placed in the location specified by the EXTERNAL clause.

For the routine to work properly, it is vital that you create it with the applicable clauses that reflect the characteristics of the routine. Many of the clauses for registering the different types of routines are common.

Prerequisites:

For the list of privileges required to create a routine in the database, see the following statements:
- CREATE FUNCTION
- CREATE METHOD
- CREATE TYPE
- CREATE PROCEDURE

Procedure:

To create 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.

To create a method, you must have have executed a CREATE TYPE statement to create a structured type. The CREATE TYPE statement contains an optional METHOD clause that can be used to optionally specify a method declaration to be associated with the type. Alternatively you can execute the ALTER TYPE statement to declare a method for an existing structured type. The method is then formally created by executing the CREATE METHOD statement. The CREATE METHOD statement only addresses attributes that relate to a method's signature.

Once you have created your routine, you can invoke it from any interface that supports routine invocation for the particular routine type. Depending on the routine type, this can include: client applications, other routines, triggers, and SQL statements.

Related concepts:
- "Routines in application development" on page 3
- "Parameter styles for external routines" on page 87
- "Performance considerations for developing routines" on page 22
- "Security considerations for routines" on page 24

Related tasks:
- "Writing routines" on page 33

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 (C)"
- "spserver.db2 -- To create a set of SQL procedures "
- "UDFsCreate.db2 -- How to catalog the UDFs contained in UDFsqlsv.java "

Writing routines

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

| There are some routine features that are specific to a routine type. 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 Types of routines (procedures, functions,
| methods).)
- 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 create 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 features that you can employ in the body of your routines 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 can 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 87
- "Performance considerations for developing routines" on page 22
- "Security considerations for routines" on page 24
- "C/C++ routines" on page 151
- "Java routines" on page 167
- "Restrictions on using routines" on page 29
- "Library and class management considerations" on page 27
- "OLE automation routine design" on page 179
- "OLE DB user-defined table functions" on page 186
- "Supported routine programming languages" on page 19

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"
- "spserver.db2 -- To create a set of SQL procedures "
- "spserver.sqc -- Definition of various types of stored procedures (C)"
- "spserver.sqC -- Definition of various 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)"

Authorizations and binding of routines that contain SQL

When discussing routine level authorization it is important to define some roles related to routines, the determination of the roles, and the privileges related to these roles:

Package Owner
The owner of a particular package that participates in the implementation of a routine. The package owner is the user who executes the BIND command to bind a package with a database, unless the OWNER precompile/BIND option is used to override the package ownership and set it to an alternate user. Upon execution of the BIND command, the package owner is granted EXECUTE WITH GRANT privilege on the package. A routine library or executable can be comprised of multiple packages and therefore can have multiple package owners associated with it.

Routine Definer

The ID that issues the CREATE statement to register a routine. The routine definer is generally a DBA, but is also often the routine package owner. When a routine is invoked, at package load time, the authorization to run the routine is checked against the definer's authorization to execute the package or packages associated with the routine (not against the authorization of the routine invoker). For a routine to be successfully invoked, the routine definer must have one of:

- EXECUTE privilege on the package or packages of the routine and EXECUTE privilege on the routine
- SYSADM or DBADM authority

If the routine definer and the routine package owner are the same user, then the routine definer will have the required EXECUTE privileges on the packages. If the definer is not the package owner, the definer must be explicitly granted EXECUTE privilege on the packages by the package owner or any user with SYSADM or DBADM authority.

Upon issuing the CREATE statement that registers the routine, the definer is implicitly granted the EXECUTE WITH GRANT OPTION privilege on the routine.

The routine definer's role is to encapsulate under one authorization ID, the privileges of running the packages associated with a routine and the privilege of granting EXECUTE privilege on the routine to PUBLIC or to specific users that need to invoke the routine.

Note: For SQL routines the routine definer is also implicitly the package owner. Therefore the definer will have EXECUTE WITH GRANT OPTION on both the routine and on the routine package upon execution of the CREATE statement for the routine.

Routine Invoker
The ID that invokes the routine. To determine which users will be invokers of a routine, it is necessary to consider how a routine can be invoked. Routines can be invoked from a command window or from within an embedded SQL application. In the case of methods and UDFs the routine reference will be embedded in another SQL statement. A procedure is invoked by using the CALL statement. For dynamic SQL in an application,

the invoker is the runtime authorization ID of the immediately higher-level routine or application containing the routine invocation (however, this ID can also depend on the DYNAMICRULES option with which the higher-level routine or application was bound). For static SQL, the invoker is the value of the OWNER precompile/BIND option of the package that contains the reference to the routine. To successfully invoke the routine, these users will require EXECUTE privilege on the routine. This privilege can be granted by any user with EXECUTE WITH GRANT OPTION privilege on the routine (this includes the routine definer unless the privilege has been explicitly revoked), SYSADM or DBADM authority by explicitly issuing a GRANT statement.

As an example, if a package associated with an application containing dynamic SQL was bound with DYNAMICRULES BIND, then its runtime 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 sufficient to execute the SQL statements in the routine body. These SQL statements might require table access privileges or execute privileges if there are 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 otherwise an error (SQLSTATE 42051) will be returned.

The steps involved in managing the execute privilege on a routine are detailed in the diagram and text that follows:

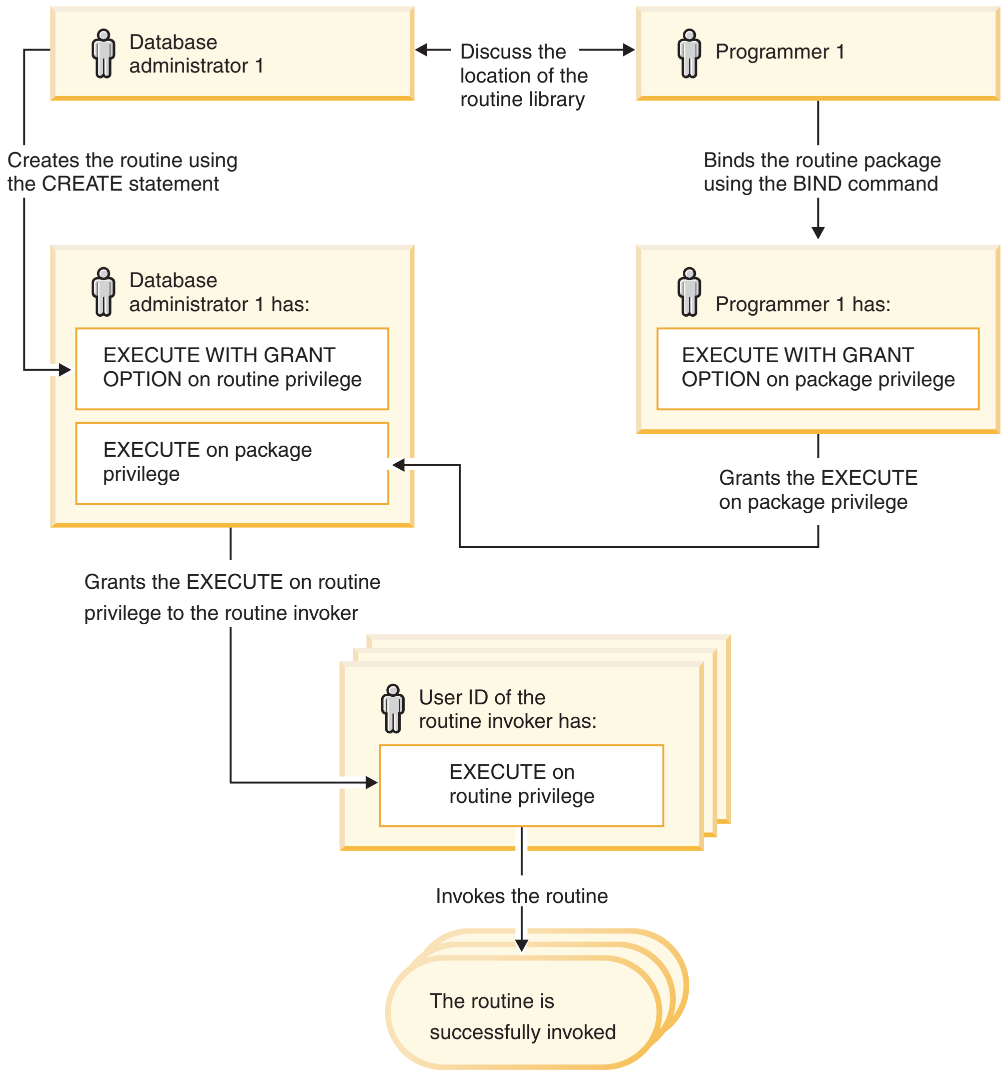

Database administrator 1
Discuss the location of the routine library
Programmer 1
Creates the routine using the CREATE statement
Binds the routine package using the BIND command
Database administrator 1 has:
EXECUTE WITH GRANT OPTION on routine privilege
EXECUTE on package privilege
Programmer 1 has:
EXECUTE WITH GRANT OPTION on package privilege
Grants the EXECUTE on package privilege
Grants the EXECUTE on routine privilege to the routine invoker
User ID of the routine invoker has:
EXECUTE on routine privilege
Invokes the routine
The routine is successfully invoked

Figure 2. Managing the EXECUTE privilege on routines

1. Definer performs the appropriate CREATE statement to register the routine. This registers the routine in DB2® with its intended level of SQL access, establishes the routine signature, and also points to the routine executable. The definer, if not also the package owner, needs to communicate with the package owners and authors of the routine programs to be clear on where the routine libraries reside so that this can be correctly specified in the EXTERNAL clause of the CREATE statement. By virtue of a successful CREATE statement, the definer has EXECUTE WITH GRANT privilege on the routine, however the definer does not yet have EXECUTE privilege on the packages of 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 program, or have it done on their behalf. Upon a successful precompile and bind, the package owner is implicitly granted EXECUTE WITH GRANT OPTION privilege on the respective package. 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 issuing of the CREATE statement for the routine.

4. Each package owner must explicitly grant EXECUTE privilege on their respective routine package to the definer of the routine. This step must come at some time after the previous step. If the package owner is also the routine definer, this step can be skipped.
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, authority levels, and database authorities" in the *Administration Guide: Implementation*
- "Effect of DYNAMICRULES bind option on dynamic SQL" on page 104
- "Routine 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*

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 operating system).

  - 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 procedures, this is: `sqllib/function` (UNIX) or `sqllib\function` (Windows). For unfenced 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.
- If you attempt to invoke a routine and receive an error (SQLCODE -551, SQLSTATE 42501) indicating that you have insufficient privileges to perform this operation, this is likely because you do not have the EXECUTE privilege on the routine. This privilege can be granted to any invoker of a routine by a user with SYSADM, DBADM authorization or by the definer of the routine. The related topic on authorizations and routines provides details on how to effectively manage the use of this privilege.

Routine debugging techniques

To debug a routine, use the following techniques:

- The Development Center provides extensive debugging tools for SQL-bodied and Java 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 procedures, a safe alternative is to write diagnostic data to an SQL table. The 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 procedure to write data (or no longer write data) to an SQL table, you must drop and re-register the procedure with (or without) the MODIFIES SQL DATA clause. Before dropping and re-registering the 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:

- "Authorizations and binding of routines that contain SQL" on page 35
- "Security considerations for routines" on page 24

Related tasks:

- "Debugging Java stored procedures" on page 175
- "Returning error messages from SQL procedures" on page 70

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 190
- "Syntax for passing arguments to routines written in C/C++, OLE, or COBOL" on page 89
- "Supported SQL data types in OLE automation" on page 182
- "Supported SQL data types in C/C++" on page 155

Data conflicts when procedures read from or write to tables

To preserve the integrity of the database, it is necessary to avoid conflicts when reading from 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 by the application. THe row is in an indeterminate state from the perspective of the routine- 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 from and writing to 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. If a procedure is invoked within a dynamic compound statement or a trigger, the compound statement or the statement that causes the firing of the trigger is the top-level statement. If an SQL function or an SQL method contains a nested CALL statement, the statement invoking the function or the method is the 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.

- A procedure is invoked from a trigger, a dynamic compound statement, an SQL function or an SQL method.

  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 accessed for reading or writing, 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 can be both read from and written to without causing a conflict.
2. If a table is being read within a table access context then other contexts can 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 can 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 in application development" on page 3
- "SQL in external routines" on page 101

Procedure features

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

Procedure parameter modes

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

There are three types of parameters for procedures:
- IN parameters: data passed to the procedure.
- OUT parameters: data returned by the procedure.
- INOUT parameters: data passed to the procedure that is, during procedure execution, replaced by data to be returned from the procedure.

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

Related concepts:
- "Procedure result sets" on page 42

Related tasks:
- "Creating routines in the database" on page 31

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

Procedure result sets

The following sections describe how with procedures you can return result sets, and shows you how to return and receive result sets using various interfaces.

Procedure result sets

In addition to exchanging parameters, 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 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 procedure, only unfetched rows are passed to the invoker as the result set. When exiting a 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 procedure with the CREATE PROCEDURE statement, indicate the number of result sets that it returns with the DYNAMIC RESULT SETS clause. This value is in the RESULT_SETS column in the SYSCAT.ROUTINES view. If the number of result sets returned from a 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 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.

| If a procedure is invoked within a trigger, a dynamic compound statement, an SQL
| function or a SQL method, any result sets will not be accessible.

Note: A COMMIT issued from within the 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 procedure, cursors can be opened and returned as result sets.

Related concepts:
- "Procedures" on page 11
- "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 procedures” on page 44
- “Receiving procedure result sets in SQL routines” on page 47
- “Receiving procedure result sets in JDBC applications and routines” on page 49
- “Returning result sets from JDBC procedures” on page 46
- “Receiving procedure result sets in SQLJ applications and routines” on page 48
- “Returning result sets from SQLJ procedures” on page 45
- “Returning result sets from CLR procedures” on page 114

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 procedures

You can develop procedures that return result sets to the invoking routine or application. In SQL and embedded SQL 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 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 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 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 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 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 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 procedure.

Related concepts:
- "Condition handlers in SQL procedures" on page 71
- "SQLCODE and SQLSTATE variables in SQL procedures" on page 74
- "Procedure result sets" on page 42

Related tasks:
- "Creating SQL procedures" in the *Application Development Guide: Building and Running Applications*
- "Calling procedures from the Command Line Processor (CLP)" on page 204
- "Calling SQL procedures with client applications" 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 procedure result sets in SQL routines" on page 47
- "Receiving procedure result sets in JDBC applications and routines" on page 49
- "Receiving procedure result sets in SQLJ applications and routines" on page 48

Related reference:
- "SQL procedure samples" in the *Application Development Guide: Building and Running Applications*

Related samples:
- "spserver.sqc -- Definition of various types of stored procedures (C)"
- "spserver.sqC -- Definition of various types of stored procedures (C++)"

Returning result sets from SQLJ procedures

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

Procedure:

To return a result set from an SQLJ 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 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 procedure.

Related concepts:

- "Procedure result sets" on page 42

Related tasks:

- "Receiving procedure result sets in SQL routines" on page 47
- "Receiving procedure result sets in JDBC applications and routines" on page 49
- "Receiving procedure result sets in SQLJ applications and routines" on page 48

Related samples:

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

Returning result sets from JDBC procedures

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

Procedure:

To return a result set from a JDBC procedure:

1. For each result set that is to be returned, include a parameter of type ResultSet[] in the 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 query 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 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:

- "Procedure result sets" on page 42

Related tasks:

- "Receiving procedure result sets in SQL routines" on page 47
- "Receiving procedure result sets in JDBC applications and routines" on page 49
- "Receiving procedure result sets in SQLJ applications and routines" on page 48

Related samples:

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

Receiving procedure result sets in SQL routines

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

Prerequisites:

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

Procedure:

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

1. DECLARE result set locators for each result set that the 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 procedure. For example:

   ```
   CALL targetProcedure();
   ```

3. ASSOCIATE the result set locator variables (defined above) with the invoked procedure. For example:
   ```
   ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
      WITH PROCEDURE targetProcedure;
   ```
4. ALLOCATE the result set cursors passed from the invoked 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:
- "Procedure result sets" on page 42

Related tasks:
- "Returning result sets from SQL and embedded SQL procedures" on page 44
- "Returning result sets from JDBC procedures" on page 46
- "Returning result sets from SQLJ procedures" on page 45

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 procedure result sets in SQLJ applications and routines

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

Procedure:

To accept 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 procedure that returns result sets. In the following example, a 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 procedure. For multiple result sets, put the getNextResultSet() call in a loop structure. Each result set returned by the 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:
- "Procedure result sets" on page 42

Related tasks:
- "Returning result sets from SQL and embedded SQL procedures" on page 44
- "Returning result sets from JDBC procedures" on page 46
- "Returning result sets from SQLJ procedures" on page 45

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

Receiving procedure result sets in JDBC applications and routines

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

Procedure:

To accept 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 procedure that returns result sets (using a CallableStatement object). In the following example, a 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 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 procedure:

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

4. Use the CallableStatement object's getResultSet() method to accept the first result set from the 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:

- "Procedure result sets" on page 42

Related tasks:

- "Returning result sets from SQL and embedded SQL procedures" on page 44
- "Returning result sets from JDBC procedures" on page 46
- "Returning result sets from SQLJ procedures" on page 45

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 procedures

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

C or C++ 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 procedure signature accepts parameters of type INTEGER, SMALLINT, and CHAR(3):

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

Java™ 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 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 procedure that accepts arguments like a main function in a C program, specify PROGRAM TYPE MAIN in the CREATE PROCEDURE statement. You must write procedures of PROGRAM TYPE MAIN to conform to the following specifications:

- The 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 procedure must be built as a shared library

In PROGRAM TYPE MAIN procedures, DB2® sets the value of the first element in the *argv* array, (*argv[0]*), to the name of the procedure. The remaining elements of the *argv* array correspond to the parameters as defined by the PARAMETER STYLE of the procedure. For example, the following embedded C 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 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 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:

- "Procedures" on page 11

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 (C)"
- "spserver.sqc -- Definition of various types of stored procedures (C)"

UDF and method features

Unlike stored procedures, UDFs and methods are invoked from within SQL statements. Whereas a stored procedure is invoked only 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 might 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 might 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 might require 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 a fixed size, the UDF or method can itself include a memory allocation and thus, can make use of 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 might 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 operating systems" on page 55
- "Method and scalar function processing model" on page 56

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 operating systems

To make your UDF or method code portable between 32-bit and 64-bit operating systems, 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 52

- “User-defined scalar functions” on page 13
- “User-defined scalar functions” on page 15

Related tasks:

- “Invoking 32-bit routines on a 64-bit database server” on page 197

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 can 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 cannot 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:

- “User-defined scalar functions” on page 13
- “Methods” on page 16

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 89

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 can be passed to the function, but they are pointing to the same values that were passed on OPEN. Therefore, the argument values might 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 can return an error.

For table UDFs not defined with FINAL CALL, only OPEN, FETCH, and CLOSE calls are made to the function. 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 can 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 59
- "User-defined scalar functions" on page 15

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.
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.

Point in scan time	**NO FINAL CALL LANGUAGE JAVA SCRATCHPAD**	**FINAL CALL LANGUAGE JAVA SCRATCHPAD**
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. The term ″UDF method″ refers to 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 333
- “Java routines” on page 167
- “Table function processing model” on page 57

Related reference:

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

Chapter 3. SQL routines

SQL routines are created by executing the appropriate CREATE statement for the routine type, in which you also specify the routine body which for an SQL routine must be entirely composed of SQL or SQL PL statements. You can use the IBM DB2 Development Center to help you create, debug, and run SQL procedures, or you can create them using the DB2 command line processor.

SQL Procedural Language (SQL PL) in DB2

Before discussing the use of SQL PL in procedures and functions it is important to first review some basic terminology and concepts related to procedureal SQL in DB2. Procedural SQL constructs such as scalar variables, IF statements and WHILE loops were introduced in DB2 with the release of DB2 Version 7. These constructs expanded into the set of SQL statements that makes up the SQL Procedural Langauge (SQL PL) that we refer to today.

SQL PL:

SQL PL, is actually a subset of SQL that provides procedural constructs that can be used to implement logic around traditional SQL statements. SQL PL is a high level programming language with a simple syntax, and common programming control statements including the IF, ELSE, WHILE, FOR, ITERATE, and GOTO statements, as well as other statements.

SQL PL and SQL procedures:

SQL PL procedures can contain parameters, variables, assignment-statements, SQL PL control statements, and compound SQL statements. SQL PL procedures also support a powerful condition and error handling mechanism, nested and recursive calls, the returning of multiple result sets to the caller or the client application. For a complete set of supported language elements in SQL PL procedures, refer to the CREATE PROCEDURE (SQL) statement in the SQL reference.

Inline SQL PL and SQL functions, triggers, and dynamic compound statements:

As of DB2 Version 7.2, a subset of SQL PL is supported in SQL functions and trigger bodies. This subset of SQL PL is known as inline SQL PL. The word inline highlights an important difference between inline SQL PL and the full SQL PL language. Whereas an SQL PL procedure is implemented by statically compiling its individual SQL queries into sections in a package, an inline SQL PL function is implemented, as the name suggests, by inlining the body of the function into the query that uses it. Some performance considerations result from this difference and should be considered when you are planning on whether to implement your procedural logic in SQL PL in a procedure or with inline SQL PL.

A dynamic-compound-statement is a statement that actually allows you to group multiple SQL statements into a small logical atomic block in which you can declare variables, and condition handling elements. These statements are compiled by DB2 as a single SQL statement and can contain elements of SQL PL. The subset of SQL PL known as inline SQL PL and only a small set of basic SQL statements can be included within a dynamic compound statement. Dynamic compound statements are useful for creating short scripts that perform small units of logical work with minimal control flow, but that have significant data flow. For more complex logic that requires parameters, passing of result sets or other more advanced procedural elements SQL procedures and functions may be more appropriate.

For a complete list of SQL PL statements that are supported in SQL PL procedures, SQL functions, and dynamic compound statements including triggers, refer to the SQL Reference CREATE statements for each routine type.

Related concepts:
- "Routines in application development" on page 3
- "User-defined routines" on page 9
- "CREATE statements for SQL routines" on page 62

Related tasks:
- "Creating SQL procedures from the command line" on page 66

CREATE statements for SQL routines

SQL routines are created by executing the appropriate CREATE statement for the routine type, in which you also specify the routine body, which for an SQL routine, must be composed only of SQL or SQL PL statements. You can use the IBM DB2 Development Center to help you create, debug, and run SQL procedures. SQL procedures, functions, and methods can also be created using the DB2 command line processor.

SQL procedures, functions, and methods each have a respective CREATE statement. The syntax for these statements is different however there are some common elements to them. In each you must specify the routine name, and parameters if there are to be any as well as a return type. You may also specify additional keywords that provide DB2 with information about the logic contained in the routine. DB2 uses the routine prototype and the additional keywords to identify the routine at invocation time, and to execute the routine with the required feature support and best performance possible.

For specific information on creating SQL procedures in the DB2 Development Center or from the Command Line Processor, or on creating functions and methods, refer to the related topics.

SQL access levels in SQL routines

An SQL access level is a clause specified in a CREATE statement for a routine that indicates the level of SQL access used in the routine. This clause is used to provide information to the database manager about the statement so that the statement can be executed safely by the database manager and with the best possible performance.

By default SQL procedures are created with SQL access level MODIFIES SQL DATA. This can be modified to a lower level of access such as READS SQL DATA or CONTAINS SQL if no table data is modified by the SQL statement within the procedure. This is done by specifying the appropriate SQL access level clause in the CREATE statement of the procedure. Optimal performance of routines is achieved when the most restrictive SQL access clause that is valid is specified in the CREATE statement.

By default all UDFs (table functions, scalar functions, methods) are created with SQL access level READS SQL DATA. The SQL access level can be defined or modified to CONTAINS SQL for SQL-bodied UDFs when no data is read within the UDF. The SQL access level for SQL bodied table functions can be defined or modified to MODIFIES SQL DATA because SQL statements that modify tables are supported within their bodies.

Related concepts:

- "SQL table functions that modify SQL data" on page 80

Related reference:

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

Dynamic SQL in SQL 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*

SQL/ SQL PL procedures

Design considerations for SQL procedures

When you are considering designing an SQL procedure you should consider the following:

Is a procedure really what I need?
Refer to the comparison of routine types to learn about how procedures and other routine types are used, and to compare the features and restrictions of each type.

Are the SQL statements that I need to execute supported within SQL procedures?
To check this refer to SQL Reference. If the statements that you require are not supported within an SQL procedure, consider implementing the logic that you require in an external procedure, SQL function, or in an application.

Would a simple dynamic compound statement be sufficient to meet my needs?
It sometimes arises that for very small and simple pieces of procedural logic, that a compound SQL statement is sufficient. Compound statements allow you to group multiple statements together as a unit to be executed together and can contain some SQL PL language elements. Compound statements are ideal if you do not require parameters, or a lot of procedural logic, but instead you require only minimal procedural logic as the logic required is intended primarily to flow data. To learn more about compound statements refer to:

- “SQL Procedural Language (SQL PL) in DB2” on page 61
- Dynamic compound SQL statement

Is an SQL function more appropriate than an SQL procedure?
If you can rewrite the SQL that you want to contain in your procedure as an expression instead of multiple SQL statements, then it is likely better for you if you implement your logic as a function because SQL expressions are more efficient and will perform better than SQL PL combined with SQL statements. For example a CASE expression will perform better than an IF ELSE or CASE statement that contains other SQL statements. For an example of an SQL procedure that was effectively rewritten as an SQL function refer to:

- “Improving the performance of SQL procedures” on page 75

* Is this procedure to be used for OLTP activities?

* If the SQL procedure you want to write is to be used in an OLTP application, it would be a good idea to read about some other DB2 features that might help you maximize the performance of your application. Some other DB2 features to discover are:

- Global temporary tables: useful for storing intermediate results, faster to access tan base tables

- Generated columns in tables: used to automatically generate column values in a row.
- RUNSTATS command: used to gather statistic about tables to improve query performance
- “Improving the performance of SQL procedures” on page 75

Performance considerations for SQL PL procedures

Performance is almost always a concern. If you are considering implementing an SQL procedure to perform complex logic, for example a complex mathematical algorithm, that will require a lot of SQL PL and very few database queries or modifications, you should consider implementing an external procedure instead. If you decide to implement an SQL PL procedure, for tips on writing SQL PL procedures that perform well, refer to:

- “Improving the performance of SQL procedures” on page 75

Related concepts:

- “Routines in application development” on page 3
- “Types of routines (procedures, functions, methods)” on page 5
- “SQL access levels in SQL routines” on page 63
- “SQL Procedural Language (SQL PL) in DB2” on page 61
- “Compound blocks and scope of variables in SQL procedures” on page 69

Creating SQL procedures from the command line

Prerequisites:

- The user must have the privileges required to execute the CREATE PROCEDURE statement for an SQL procedure.
- Privileges to execute all of the SQL statements included within the SQL-procedure-body of the procedure.
- Any database objects referenced in the CREATE PROCEDURE statement for the SQL procedure must exist prior to the execution of the statement. execution

Procedure:

- Select an alternate terminating character for the Command Line Processor (DB2 CLP), other than the default terminating character which is a semicolon (';'), to use in the script that you will prepare in the next step.

  This is required so that the CLP can distinguish the end of SQL statements that appear within the body of a routine’s CREATE statement from the end of the CREATE PROCEDURE statement itself. The semicolon character must be used to terminate SQL statements within the SQL routine body and the chosen alternate terminating character should be used to terminate the CREATE statement and any other SQL statements that you might contain within your CLP script.

  For example, in the following CREATE PROCEDURE statement, the 'at;' sign ('@') is used as the terminating character for a DB2 CLP script named myCLPscript.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
  @
```

- Run the DB2 CLP script containing the CREATE PROCEDURE statement for the procedure from the command line, using the following CLP command:

  ```
  db2 -td <terminating-character> -vf <CLP-script-name>
  ```

  where <*terminating-character*> is the terminating character used in the CLP script file *CLP-script-name* that is to be run.

  The DB2 CLP option -td indicates that the CLP terminator default is to be reset with *terminating character*. The *-vf* indicates that the CLP's optional verbose (*-v*) option is to be used which will cause each SQL statement or command in the script to be displayed to the screen as it is run, along with any output that results from its execution. The *-f* option indicates that the target of the command is a file.

  To run the specific script shown in the first step, issue the following command from the system command prompt:

  ```
  db2 -td@ -vf myCLPscript.db2
  ```

Related concepts:

- "Routines in application development" 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*

Parameters in SQL procedures

DB2 supports the use of input, output, and input and output parameters in SQL procedures. The keywords IN, OUT, and INOUT in the CREATE PROCEDURE statement indicate the mode or intended use of the parameter. IN and OUT parameters are passed by value, and INOUT parameters are passed by reference.

Parameters are optional and you can create SQL procedures that do not have any, however, when multiple parameters are specified they must be unique within the procedure. Also if a variable is to be declared within the procedure with the same name as a parameter, it must be declared within a labeled atomic block nested within the procedure otherwise DB2 will detect what would otherwise be an ambiguous reference.

Parameters to SQL procedures cannot be named either of SQLSTATE or SQLCODE regardless of the data type for the parameter. See the CREATE PROCEDURE statement for complete details about parameter modes and restrictions on parameters in SQL procedures.

The following SQL procedure named *myparams* illustrates the use of IN, INOUT, and OUT parameter modes. Let us say that SQL procedure is defined in a CLP file named *myfile.db2* and that we are using the command line.

```
CREATE PROCEDURE myparams (IN p1 INT, INOUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
 SET p2 = p2 + 1;
 SET p3 = 2 * p1;
END@
```

To create the stored procedure, from the command line enter the following:

```
db2 -td@ -vf myfile.db2
```

Then to call the procedure, from the command line enter the following:

```
db2 "CALL myParms(1, 3, ?)"
```

The '?' is a parameter marker for the output parameter. You must specify an input value for any INOUT parameters, even if they are not referenced within the procedure. The following output will be returned:

```
  Value of output parameters
 --------------------------
 Parameter Name  : P2
 Parameter Value : 4

 Parameter Name  : P3
 Parameter Value : 2

 Return Status = 0
```

Related concepts:

- "Parameter styles for external routines" on page 87
- "Procedure parameter modes" on page 42
- "SQL Procedural Language (SQL PL) in DB2" on page 61
- "CREATE statements for SQL routines" on page 62
- "Variables in SQL procedures (DECLARE, DEFAULT, SET statements)" on page 68

Related tasks:

- "Creating SQL procedures from the command line" on page 66

Variables in SQL procedures (DECLARE, DEFAULT, SET statements)

Variables are supported in procedures as part of a compound statement. The keyword DECLARE is used when declaring variables. The variable declarations must appear at the top of the SQL procedure body, before the declaration of any conditions, condition handlers, cursors or any SQL statements.

Variables can be declared with a default value by using the DEFAULT clause where the default value can be a constant, special register value, or an expression. For example:

```
 CREATE PROCEDURE P2(INOUT a VARCHAR(8),
                     OUT  b INTEGER)
LANGUAGE SQL
```

```
 BEGIN
   DECLARE var1 INTEGER DEFAULT 0;
   DECLARE var2 VARCHAR(5) DEFAULT  a || 'bc';

    -- other SQL statements --

  END@
```

Below the variable declarations of an SQL procedure, the variables and parameters, including input parameters, can be assigned values by using the assignment-statement as follows:

```
  CREATE PROCEDURE P2(INOUT a VARCHAR(8),
                      OUT  b INTEGER)
 LANGUAGE SQL
 BEGIN
   DECLARE var1 INTEGER DEFAULT 0;
   DECLARE var2 VARCHAR(5) DEFAULT  a || 'bc';

    SET var1 = 0;
    SET var1 = var1 + 1;
    SET var2  = var2 || 'def';

    SET a = var1;
    SET b = var2;
  END@
```

Related concepts:

- "SQL Procedural Language (SQL PL) in DB2" on page 61
- "CREATE statements for SQL routines" on page 62
- "Parameters in SQL procedures" on page 67
- "Improving the performance of SQL procedures" on page 75

Related tasks:

- "Creating SQL procedures from the command line" on page 66

Compound blocks and scope of variables in SQL procedures

Within an SQL procedure you can have one or more compound statements. Compound statements introduce a block of SQL statements that are compiled and executed as single statement in DB2. Compound statements are easily recognized as starting and ending with the keywords BEGIN and END and can be labeled to identify the code block.

The use of a label becomes important in the context of scope of variables as it can be used to qualify the names of variables which is important in the identification and referencing of variables in different compound statements or in nested compound statements.

In the following example there are two declarations of the variable *a*. One instance of it is declared in the outer compound statement that is labelled by *lab1*, and the second instance is declared in the inner compound statement labelled by *lab2*. As it is written, DB2 will presume that the reference to *a* in the assignment-statement is the one which is in the local scope of the compound block, labelled by *lab2*. However, if the intended instance of the variable *a* was the one declared in the compound statement block labeled with ″lab1″, then to correctly reference it in the innermost compound block, the variable should have been qualified with the label of that block. That is, qualified as: *lab1.a*.

```
CREATE PROCEDURE P1 ()
LANGUAGE SQL
  lab1: BEGIN
    DECLARE a INT DEFAULT 100;
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

      UPDATE T1
       SET T1.b = 5
        WHERE T1.b = a;  <-- Variable a refers to lab2.a
                             unless qualified otherwise

    lab2: END;
 END lab1@
```

The outermost compound statement in an SQL procedure can be declared to be atomic, by adding the keyword ATOMIC after the BEGIN keyword. If any error occurs in the execution of the statements that comprise the atomic compound statement, then the entire compound statement is rolled back.

Related concepts:
- "SQL Procedural Language (SQL PL) in DB2" on page 61
- "Parameters in SQL procedures" on page 67
- "Variables in SQL procedures (DECLARE, DEFAULT, SET statements)" on page 68

Related tasks:
- "Creating SQL procedures from the command line" on page 66

Returning error messages from SQL procedures

When you issue a CREATE PROCEDURE statement for an SQL procedure, DB2 might 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:

- "Debugging routines" on page 38

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 74
- "Condition handler declarations" on page 71
- "SQLCODE and SQLSTATE variables in SQL procedures" on page 74

Related tasks:

- "Returning error messages from SQL procedures" on page 70

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 71
- "SIGNAL and RESIGNAL statements in condition handlers" on page 74
- "SQLCODE and SQLSTATE variables in SQL procedures" on page 74

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 71
- "Condition handler declarations" on page 71
- "SQLCODE and SQLSTATE variables in SQL procedures" on page 74

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 71
- "SIGNAL and RESIGNAL statements in condition handlers" on page 74
- "Condition handler declarations" on page 71

Improving the performance of SQL procedures

Overview of how DB2 compiles SQL PL and inline SQL PL:

Before discussing how to improve the performance of SQL procedures we should discuss how DB2 compiles them upon the execution of the CREATE PROCEDURE statement.

When an SQL procedure is created, DB2 separates the SQL queries in the procedure body from the procedural logic. To maximize performance, the SQL queries are statically compiled into sections in a package. For a statically compiled query, a section consists mainly of the access plan selected by the DB2 optimizer for that query. A package is a collection of sections. For more information on packages and sections, please refer to the DB2 SQL Reference. The procedural logic is compiled into a dynamically linked library.

During the execution of a procedure, every time control flows from the procedural logic to an SQL statement, there is a "context switch" between the DLL and the DB2 engine. As of DB2 Version 8.1, SQL procedures run in "unfenced mode". That is they run in the same addressing space as the DB2 engine. Therefore the context switch we refer to here is not a full context switch at the operating system level, but rather a change of layer within DB2. Reducing the number of context switches in procedures that are invoked very often, such as procedures in an OLTP application, or that process large numbers of rows, such as procedures that perform data cleansing, can have a noticeable impact on their performance.

Whereas an SQL procedure containing SQL PL is implemented by statically compiling its individual SQL queries into sections in a package, an inline SQL PL function is implemented, as the name suggests, by inlining the body of the function into the query that uses it. Queries in SQL functions are compiled together, as if the function body were a single query. The compilation occurs every time a statement that uses the function is compiled. Unlike what happens in SQL procedures, procedural statements in SQL functions however are not executed in a different layer than dataflow statements. Therefore, there is no context switch every time control flows from a procedural to a dataflow statement or vice versa.

If there are no side-effects in your logic use an SQL function instead:

Because of the difference in compilation between SQL PL in procedures and inline SQL PL in functions, it is reasonable to presume that a piece of procedural code will execute faster in a function than in a procedure if it only queries SQL data and does no data modifications - that is it has no side-effects on the data in the database or external to the database.

That is only good news if all the statements that you need to execute are supported in SQL functions. SQL functions can not contain SQL statements that modify the database. As well, only a subset of SQL PL is available in the inline SQL PL of functions. For example: you cannot execute CALL statements, declare cursors, or return result sets in SQL functions.

Here is an example of an SQL procedure containing SQL PL that was a good candidate for conversion to an SQL function to maximize performance:

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                           IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor eq; ssq;Vendor 1ssq;
    THEN SET price eq; (SELECT ProdPrice
                        FROM V1Table
                        WHERE Id = Pid);
  ELSE IF Vendor eq; ssq;Vendor 2ssq;
    THEN SET price eq; (SELECT Price FROM V2Table
                        WHERE Pid eq; GetPrice.Pid);
  END IF;
END
```

Here is the rewritten SQL function:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), PId INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
  DECLARE price DECIMAL(10,3);
  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice
                         FROM V1Table
                         WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
               WHERE Pid = GetPrice.Pid);
  END IF;
  RETURN price;
END
```

Remember that the invocation of a function is different than a procedure. To invoke the function, use the VALUES statement or invoke the function where an expression is valid, such as in a SELECT or SET statement. Any of the following are valid ways of invoking the new function:

```
VALUES (GetPrice('IBM', 324))

SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

SET price = GetPrice(Vname, Pid)
```

Avoid multiple statements in an SQL PL procedure when just one would is sufficient:

Although it is generally a good idea to write concise SQL, it is very ease to forget to do this in practice. For example the following SQL statements:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1);
INSERT INTO tab_comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

can be rewritten as a single statement:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

The multi-row insert will require roughly one third of the time required to execute the three original statements. Isolated, this improvement might seem negligible, but if the code fragment is executed repeatedly, for example, in a loop or in a trigger body, the improvement can be significant.

Similarly, a sequence of SET statements like:

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

can be written as a single VALUES statement:

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

This transformation preserves the semantics of the original sequence if there are no dependencies between any two statements. To illustrate this, consider:

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

Converting the previous two statements to:

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

does not preserve the original semantics because the expressions before the INTO keyword are evaluated 'in parallel'. This means that the value assigned to *B* is not based on the value assigned to *A*, which was the intended semantics of the original statements.

Reduce multiple SQL statements to a single SQL expression:

Like other programming languages, the SQL language provides two types of conditional constructs: procedural (IF and CASE statements) and functional (CASE expressions). In most circumstances where either type can be used to express a computation, using one or the other is a matter of taste. However, logic written using CASE expressions is not only more compact, but also more efficient than logic written using CASE or IF statements.

Consider the following fragment of SQL PL code:

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price)semi;
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice)semi;
END IF;
```

The condition in the IF clause is only being used to decide what value is inserted in the tab_comp.Val column. To avoid the context switch between the procedural and the dataflow layers, the same logic can be expressed as a single INSERT with a CASE expression:

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
            END);
```

It’s worth noting that CASE expressions can be used in any context where a scalar value is expected. In particular, they can be used on the right-hand side of assignments. For example:

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

can be rewritten as:

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

In fact, this particular example admits an even better solution:

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

Don’t underestimate the benefit in taking the time to analyze your SQL and rewriting it if required. The performance benefits will pay you back many times over for the time invested analyzing and rewriting your procedure.

Exploit the set-at-a-time semantics of SQL:

Procedural constructs such as loops, assignment and cursors allow us to express computations that would not be possible to express using just SQL DML statements. But when we have procedural statements at our disposal, there is a risk that we could turn to them even when the computation at hand can, in fact, be expressed using just SQL DML statements. As we’ve mentioned earlier, the performance of a procedural computation can be orders of magnitude slower than the performance of an equivalent computation expressed using DML statements. Consider the following fragment of code:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

To begin with, the loop body can be improved by applying the transformation discussed in the last section - "Reduce multiple SQL statements to a single SQL expression":

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

But upon closer inspection, the whole block of code can be written as an INSERT with a sub-SELECT:

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

In the original formulation, there was a context switch between the procedural and the dataflow layers for each row in the SELECT statements. In the last formulation, there is no context switch at all, and the optimizer has a chance to globally optimize the full computation.

On the other hand, this dramatic simplification would not have been possible if each of the INSERT statements targeted a different table, as shown below:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

However, the set-at-a-time nature of SQL can also be exploited here:

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
```

```
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

When looking at improving the performance of existing procedural logic, any time spent in eliminating cursor loops will likely pay off.

Keep the DB2 optimizer informed:

When a procedure is created, its individual SQL queries are compiled into sections in a package. The DB2 optimizer chooses an execution plan for a query based, among other things, on table statistics (for example, table sizes or the relative frequency of data values in a column) and indexes available at the time the query is compiled. When tables suffer significant changes, it may be a good idea to let DB2 collect statistics on these tables again. And when statistics are updated, or when new indexes are created it may also be a good idea to rebind the packages associated with SQL procedures that use the tables, to let DB2 create plans that exploit the latest statistics and indexes.

Table statistics can be updated using the RUNSTATS command. To rebind the package associated with an SQL procedure, you can use the REBIND_ROUTINE_PACKAGE built-in procedure that is available in DB2 Version 8.1. For example, the following command can be used to rebind the package for procedure MYSCHEMA.MYPROC:

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

where 'P' indicates that the package corresponds to a procedure and ′ANY′ indicates that any of the functions and types in the SQL path are considered for function and type resolution. See the Command Reference entry for the REBIND command for more details.

Related concepts:

- "Types of routines (procedures, functions, methods)" on page 5
- "SQL Procedural Language (SQL PL) in DB2" on page 61
- "Design considerations for SQL procedures" on page 65

Related tasks:

- "Creating SQL procedures from the command line" on page 66

SQL table functions

SQL table functions that modify SQL data

When the MODIFIES SQL DATA clause is specified in the CREATE FUNCTION statement of an SQL table function, the body of the SQL table function can include SQL statements that modify table data. All statements supported in a dynamic-compound statement are supported in the body of an SQL table function including:

- INSERT
- UPDATE
- DELETE
- MERGE
- SELECT where the FROM clause references a data change statement

An SQL table function that modifies SQL data is a useful way to encapsulate work that modifies table data and return a result set. The result set can be used to return rows that were accessed or modified within the table function. Modified rows of multiple tables can be returned in a single result set. An SQL table function that modifies SQL data can be used to audit transactions that access or modify table data.

Support of the MODIFIES SQL DATA clause is limited to SQL procedures and functions. You can not create an external table function specifying MODIFIES SQL DATA.

An SQL Table function that modifies SQL data:
- can only be referenced in the outermost FROM clause of a fullselect embedded in a SELECT, SELECT INTO, SET, or RETURN statement.
- must be the only SQL Table function that modifies SQL data within a FROM clause of a SELECT statement.
- must appear as the last table reference in the FROM clause when there are multiple table references
- must correlate to all other table references in the FROM clause
- can not be referenced in the body of a view definition
- can be nested within a routine if the routine modifies SQL data, or within an after-trigger

As with all routines, a table function can only be successfully invoked if the definer of the table function is authorized to execute all of the SQL statements in the body of the function.

These restrictions ensure a deterministic evaluation of the table functions and tables in the statement. The table references preceding an SQL table function will be entirely evaluated before the SQL table function gets executed. Table references in the SELECT list or WHERE clause of the SELECT statement will be evaluated after the SQL table function execution completes.

Examples of SQL Table Functions that Modify SQL Data:

Note: To see the complete prerequisite SQL associated with these examples or to run a related SQL sample, refer to sample tbfnuse.db2 and prerequisite script tbfn.db2.

Example 1: An SQL table function that modifies SQL data:

This table function updates the quantity of an item in an inventory table. An UPDATE statement is used to update the quantity of the item specified by `itemNo` in table `Inventory`, by the amount specified by `amount`. A result set containing the product name and the new quantity of the item is returned. Note that the `MODIFIES SQL DATA` clause is used because function updates table data.

```
CREATE FUNCTION updateInv(itemNo VARCHAR(20), amount INTEGER)
   RETURNS TABLE (productName varchar(20),
                  quantity INTEGER)
   LANGUAGE SQL
   MODIFIES SQL DATA
   BEGIN ATOMIC

     UPDATE Inventory as I
       SET quantity = quantity + amount
         WHERE I.itemID = itemNo;
     RETURN
       SELECT I.itemName, I.quantity
         FROM Inventory as I
           WHERE I.itemID = itemNo;
   END
```

Example 2: Invocation of an SQL table function that modifies SQL data:

The SQL table function of Example 1 is invoked from a SELECT statement. The quantity of an item identified by item number, 'ISBN-0-8021-3424-6' is increased by five (5). The product name and the updated quantity for the item are returned.

```
SELECT productName, quantity
  FROM TABLE(updateInv('ISBN-0-8021-3424-6', 5)) AS T


PRODUCTNAME          QUANTITY
-------------------- -----------
Feng Shui at Home             15
```

Example 3: Invoking an SQL table function that modifies SQL data which is correlated to another table-reference:

In this example, the quantities of multiple existing items in the inventory table 'Inventory' are updated. The VALUES clause is used to generate table-reference, 'newItem', which contains rows of items to be updated. The table function 'updateInv' is correlated to table reference 'newItem', because at least one column in 'newItem' appears as an argument to the 'updateInv' table function. Note that the table function is the last table reference in the FROM clause.

```
SELECT newItem.id, TF.productName, TF.quantity
     FROM (VALUES ('ISBN-0-8021-3424-6', 2),
                  ('ISBN-0-8021-4612-1', 5)) AS newItem(id, quantity),
           TABLE(updateInv(newItem.id, newItem.quantity)) AS TF


ID                 PRODUCTNAME          QUANTITY
------------------ -------------------- -----------
ISBN-0-8021-3424-6 Feng Shui at Home             12
ISBN-0-8021-4612-1 Baseball Heroes               15
```

To express more complex queries that reference SQL table functions in subselects or that would require multiple table references in the FROM clause, you can use common table expressions. Using a common table expression is a practical way of isolating the SQL table function in the outermost select and for ensuring that the table function modifying SQL data is the last table-reference in the FROM clause.

Example 4: Invocation of an SQL table function that modifies SQL data which is correlated to another table-reference and in a common-table-expression:

This example extends example 3 by returning the unit price and total inventory value of the updated stock items. The total inventory value is calculated by multiplying the new quantities of these items by the price from a price list table, `priceList`.

```
WITH newInv(itemNo, quantity) AS
  (SELECT id, TF.quantity
     FROM (VALUES ('ISBN-0-8021-3424-6', 5),
                  ('ISBN-0-8021-4612-1', 10)) AS newItem(id, q),
           TABLE(updateInv(newItem.id, newItem.q)) AS TF)
SELECT itemNo, quantity, unitPrice, (quantity * unitPrice) as TotalInvValue
  FROM newInv, priceList
```

```
    WHERE itemNo = priceList.itemID


ITEMNO            QUANTITY    UNITPRICE TOTALINVVALUE
----------------- ----------- --------- -----------------
ISBN-0-8021-3424-6          12     10.00           120.00
ISBN-0-8021-4612-1          15     20.00           300.70
```

Related tasks:

- "Auditing using SQL table functions" on page 83

Related reference:

- "CREATE FUNCTION (SQL Scalar, Table, or Row) statement" in the *SQL Reference, Volume 2*
- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Supported SQL statements" in the *SQL Reference, Volume 2*

Auditing using SQL table functions

Database administrators interested in monitoring table data accesses and table data modifications made by database users can audit transactions on a table by creating and using SQL table functions that modify SQL data.

Any table function that encapsulates SQL statements that perform a business task, such as updating an employee'/s personal information can additionally include SQL statements that record, in a separate table, details about the table accesses or modifications made by the user that invoked the function. An SQL table function can even be written so that it returns a result set of table rows that were accessed or modified in the body of the table function. The returned result set rows can be inserted into and stored in a separate table as a history of the changes made to the table.

Prerequisites:

For the list of privileges required to create and register an SQL table function, see the following statements:

- CREATE FUNCTION (SQL Scalar, Table, or Row) statement

The definer of the SQL table function must also have authority to run the SQL statements encapsulated in the SQL table function body. Refer to the list of privileges required for each encapsulated SQL statement. To grant INSERT, UPDATE, DELETE privileges on a table to a user, see the following statement:

- GRANT (Table, View, or Nickname Privileges) statement

The tables accessed by the SQL table function must exist prior to invocation of the SQL table function.

Example 1: Auditing accesses of table data using an SQL table function:

This function accesses the salary data of all employees in a department specified by input argument `deptno`. It also records in an audit table, named `audit_table`, the user ID that invoked the function, the name of the table that was read from, a description of what information was accessed, and the current time. Note that the table function is created with the keywords MODIFIES SQL DATA because it contains an INSERT statement that modifies SQL data.

```
CREATE FUNCTION sal_by_dept (deptno CHAR(3))
  RETURNS TABLE (lastname VARCHAR(10),
                 firstname VARCHAR(10),
                 salary INTEGER)
  LANGUAGE SQL
  MODIFIES SQL DATA
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  BEGIN ATOMIC
    INSERT INTO audit_table(user, table, action, time)
      VALUES (USER,
              'EMPLOYEE',
              'Read employee salaries in department: ' || deptno,
              CURRENT_TIMESTAMP);
    RETURN
      SELECT lastname, firstname, salary
        FROM employee as E
          WHERE E.dept = deptno;
  END
```

Example 2: Auditing updates to table data using an SQL table function:

This function updates the salary of an employee specified by `updEmpNum`, by the amount specified by `amount`, and also records in an audit table named `audit_table`, the user that invoked the routine, the name of the table that was modified, and the type of modification made by the user. A SELECT statement that references a data change statement (here an UPDATE statement) in the FROM clause is used to return the updated row values. Note that the table function is created with the keywords MODIFIES SQL DATA because it contains both an INSERT statement and a SELECT statement that references the data change statement, UPDATE.

```
CREATE FUNCTION update_salary(updEmpNum CHAR(4), amount INTEGER)
  RETURNS TABLE (emp_lastname VARCHAR(10),
                 emp_firstname VARCHAR(10),
                 newSalary INTEGER)
  LANGUAGE SQL
  MODIFIES SQL DATA
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  BEGIN ATOMIC
    INSERT INTO audit_table(user, table, action, time)
    VALUES (USER,
            'EMPLOYEE',
            'Update emp salary. Values: '
               || updEmpNum || ' ' || char(amount),
            CURRENT_TIMESTAMP);
    RETURN
      SELECT lastname, firstname, salary
        FROM FINAL TABLE(UPDATE employee
                         SET salary = salary + amount
                         WHERE employee.empnum = updEmpNum);
  END
```

Example 3: Invoking an SQL table function used for auditing transactions:

The following shows how a user might invoke the routine to update an employee's salary by 500 yen:

```
SELECT emp_lastname, emp_firstname, newsalary
  FROM TABLE(update_salary(CHAR('1136'), 500)) AS T
```

A result set is returned with the last name, first name, and new salary for the employee. The invoker of the function will not know that the audit record was made.

```
EMP_LASTNAME EMP_FIRSTNAME NEWSALARY
------------ ------------- -----------
JONES        GWYNETH             90500
```

The audit table would include a new record such as the following:

```
USER     TABLE      ACTION                              TIME
-------- ---------- ----------------------------------  --------------------------
MBROOKS  EMPLOYEE   Update emp salary. Values: 1136 500  2003-07-24-21.01.38.459255
```

Example 4: Retrieving rows modified within the body of an SQL table function:

This function updates the salary of an employee, specified by an employee number EMPNUM, by an amount specified by `amount`, and returns the original values of the modified row or rows to the caller. This example makes use of a SELECT statement that references a data change statement in the FROM clause. Specifying OLD TABLE within the FROM clause of this statement flags the return of the original row data from the table `employee` that was the target of the UPDATE statement. Using FINAL TABLE, instead of OLD TABLE, would flag the return of the row values subsequent to the update of table employee.

```
CREATE FUNCTION update_salary (updEmpNum CHAR(4), amount DOUBLE)
  RETURNS TABLE (empnum CHAR(4),
               emp_lastname  VARCHAR(10),
               emp_firstname VARCHAR(10),
               dept CHAR(4),
               newsalary integer)
  LANGUAGE SQL
  MODIFIES SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  BEGIN ATOMIC
    RETURN
      SELECT empnum, lastname, firstname, dept, salary
        FROM OLD TABLE(UPDATE employee
                              SET salary = salary + amount
                                WHERE employee.empnum = updEmpNum);
  END
```

Related concepts:

- "SQL table functions that modify SQL data" on page 80

Related reference:

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

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 • .NET common language runtime languages • 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 • .NET common language runtime languages • 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.
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++ • .NET common language runtime languages • 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 z/OS and OS/390.

Table 1. Parameter styles (continued)

Parameter style	Supported language	Supported routine type	Description
GENERAL WITH NULLS	• C/C++ • .NET common language runtime languages • 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 z/OS and OS/390.

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.
3. .NET common language runtime methods are not supported.<

Related concepts:

- "DB2GENERAL routines" on page 333
- "Java routines" on page 167

Related reference:

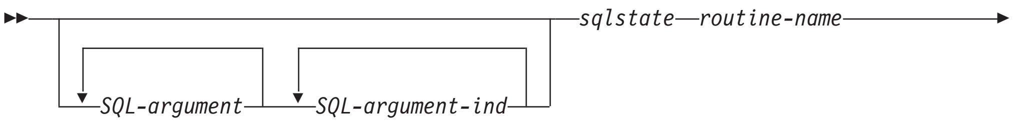
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in the *SQL Reference, Volume 2*
- "CREATE METHOD statement" in the *SQL Reference, Volume 2*
- "Syntax for passing arguments to routines written in C/C++, OLE, or COBOL" on page 89

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.

PARAMETER STYLE SQL routines



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

PARAMETER STYLE DB2SQL procedures

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

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

PARAMETER STYLE GENERAL procedures

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

PARAMETER STYLE GENERAL WITH NULLS procedures

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

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 operating system, also known as 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	The result is not NULL.
-1	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 operating system.

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 operating system.

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 operating system.

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 operating system.

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 operating system.

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 truncates 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 routine. 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 operating system.

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 will 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 can 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 can 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. The table function can 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 can 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 operating system.

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 might 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 *database 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 *database 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. Environment code pages (codepg)

   This is a union of three 48-byte structures; one is common to all DB2 Universal Database products (cdpg_db2), one is used by routines written for older versions of DB2 Universal Database (cdpg_cs), and the last is for use by older versions of DB2 UDB for z/OS and OS/390 (cdpg_mvs). For portability, it is recommended that the common structure, cdpg_db2, be used in all routines.

   The cdgp_db2 structure is made up of an array (db2_ccsids_triplet) of three sets of code page information representing the possible encoding schemes in the database as follows:

   a. ASCII encoding scheme. Note that for compatibility with previous version of DB2 Universal Database, if the database is a Unicode database then the information for the Unicode encoding scheme will be placed here as well as appearing in the third element.

b. EBCDIC encoding scheme

c. Unicode encoding scheme

Following the encoding scheme information is the array index of the encoding scheme for the routine (db2_encoding_scheme).

Each element of the array is composed of three fields:
- db2_sbcs. Single byte code page, an unsigned long integer.
- db2_dbcs. Double byte code page, an unsigned long integer.
- db2_mixed. Composite code page (also called mixed 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-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 system (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 operating system or platform

    For additional operating systems 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* can be equal to 0, and *n* 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 can 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 can 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 87
- "Include file for C/C++ routines (sqludf.h)" on page 154
- "C/C++ routines" on page 151

Related tasks:

- "Writing routines" on page 33

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*
- "appl_id - Application ID monitor element" 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, Visual Basic, C#, Java™, and others) can contain SQL.

The CREATE statement for a routine (stored procedure, UDF), or the CREATE TYPE statement, in the case of a method, contains a clause that defines the level of SQL access for the routine or method. 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 can 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 and SQL-bodied table functions.

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 because it is attempting a read of SQL data. Also note that nested routine references, must be of the same or of a more restrictive SQL level that contains the reference. For example, routines that modify SQL data can invoke routines that read SQL data, but routines that can only read SQL data, that are defined with the READS SQL DATA clause, cannot invoke routines that modify SQL data.

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 returns.

A routine can 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 can instead 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 can 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*

Effect of DYNAMICRULES bind option 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

Related concepts:

- "Authorization Considerations for Dynamic SQL" in the *Application Development Guide: Programming Client Applications*
- "Authorizations and binding of routines that contain SQL" on page 35

.NET common language runtime routines

The following sections describe how to write .NET routines to be executed by the .NET Framework common language runtime.

Common language runtime (CLR) routines

In DB2®, a common language runtime (CLR) routine is an external routine created by executing a CREATE PROCEDURE or CREATE FUNCTION statement that references a .NET assembly as its external code body.

The following terms are important in the context of CLR routines:

.NET Framework
: A Microsoft® application development environment comprised of the CLR and .NET Framework class library designed to provide a consistent programming environment for developing and integrating code pieces.

Common language runtime (CLR)
: The runtime interpreter for all .NET Framework applications.

intermediate language (IL)
: Type of compiled byte-code interpreted by the .NET Framework CLR. Source code from all .NET compatible languages compiles to IL byte-code.

assembly
: A file that contains IL byte-code. This can either be a library or an executable.

You can implement CLR routines in any language that can be compiled into an IL assembly. These languages include, but are not limited to: Managed C++, C#, Visual Basic, and J#.

Before developing a CLR routine, it is important to both understand the basics of routines and the unique features and characteristics specific to CLR routines. To learn more about routines and CLR routines see:
- "Routines in application development" on page 3
- "Supported SQL data types for the DB2 .NET Data Provider" on page 110
- "Parameters in CLR routines" on page 111
- "Returning result sets from CLR procedures" on page 114
- "Restrictions on CLR routines" on page 116
- "Errors related to CLR routines" on page 117

Developing a CLR routine is easy. For step-by-step instructions on how to develop a CLR routine and complete examples see:
- "Creating CLR routines"
- "Examples of CLR procedures in C#" on page 119
- "Examples of CLR user-defined functions in C#" on page 139

Related concepts:
- "Routines in application development" on page 3
- "Parameter styles for external routines" on page 87
- "SQL in external routines" on page 101
- "Types of routines (procedures, functions, methods)" on page 5
- "Authorizations and binding of routines that contain SQL" on page 35

Related tasks:
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running Applications*

Related samples:
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.cs"
- "SpServer.cs -- C# external code implementation of procedures created in spcat.db2"
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.vb"
- "SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2"

Creating CLR routines

Procedures and functions that reference an IL assembly are created in the same way as any external routine is created. You would choose to implement an external routine in a .NET language if:
- You want to encapsulate complex logic into a routine that accesses the database or that performs an action outside of the database.
- You require the encapsulated logic to be invoked from any of: multiple applications, the CLP, another routine (procedure, function (UDF), or method), or a trigger.
- You are most comfortable coding this logic in a .NET language.

Prerequisites:

- Knowledge of CLR routine implementation. To learn about CLR routines in general and about CLR features, see:
  - "Common language runtime (CLR) routines" on page 106
- The database server must be running a Windows operating system that supports the Microsoft .NET Framework.
- The .NET Framework, version 1.1, must be installed on the server. The .NET Framework is independently available or as part of the Microsoft .NET Framework 1.1 Software Development Kit.
- The following versions of DB2 must be installed:
  - Server: DB2 8.2 or a later release.
  - Client: Any client that can attach to a DB2 8.2 instance will be able to invoke a CLR routine. It is recommended that you install DB2 Version 7.2 or a later release on the client.
- Authority to execute the CREATE statement for the external routine. For the privileges required to execute the CREATE PROCEDURE statement or CREATE FUNCTION statement, see the details of the appropriate statement.

Restrictions:

For a list of restrictions associated with CLR routines see:

- "Restrictions on CLR routines" on page 116

Procedure:

1. Code the routine logic in any CLR supported language.
   - For general information about .NET CLR routines and .NET CLR routine features see the topics referenced in the Prerequisites section
   - Use or import `IBM.Data.DB2` if your routine will execute SQL.
   - Declare host variables and parameters correctly using data types that map to DB2 SQL data types. For a datatype mapping between DB2 and .NET datatypes:
     - "Supported SQL data types for the DB2 .NET Data Provider" on page 110
   - Parameters and parameter null indicators must be declared using one of DB2's supported parameter styles and according to the parameter requirements for .NET CLR routines. As well, scratchpads for UDFs, and the DBINFO class are passed into CLR routines as parameters. For more on parameters and prototype declarations see:
     - "Parameters in CLR routines" on page 111
   - If the routine is a procedure and you want to return a result set to the caller of the routine, you do not require any parameters for the result set. For more on returning result sets from CLR routines:
     - "Returning result sets from CLR procedures" on page 114
   - Set a routine return value if required. CLR scalar functions require that a return value is set before returning. CLR table functions require that a return code is specified as an output parameter for each invocation of the table function. CLR procedures do not return with a return value.
2. Build your code into an intermediate language (IL) assembly to be executed by the CLR. For information on how to build CLR .NET routines that access DB2, see the related link:
   - Building common language runtime routines

3. Copy the assembly into the DB2 *function directory* on the database server. It is recommended that you store assemblies or libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the assembly to another directory on the server if you wish, but to successfully invoke the routine you must note the fully qualified path name of your assembly as you will require it for the next step.
4. Execute either dynamically or statically the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION.
   - Specify the `LANGUAGE` clause with value: `CLR`.
   - Specify the `PARAMETER STYLE` clause with the name of the supported parameter style that was implemented in the routine code.
   - Specify the `EXTERNAL` clause with the name of the assembly to be associated with the routine using one of the following values:
     - the fully qualified path name of the routine assembly.
     - the relative path name of the routine assembly relative to the function directory.

     By default DB2 will look for the assembly by name in the function directory unless a fully qualified or relative path name for the library is specified in the EXTERNAL clause.

     When the CREATE statement is executed, if the assembly specified in the EXTERNAL clause is not found by DB2 you will receive an error (SQLCODE -20282) with reason code 1.
   - Specify DYNAMIC RESULT SETS with value 1 if your routine is a procedure and it will return a result set to the caller.
   - You can not specify the NOT FENCED clause for CLR procedures. By default CLR procedures are executed as FENCED procedures.

To invoke your CLR routine, see "Routine invocation" on page 193

Related concepts:
- "Common language runtime (CLR) routines" on page 106
- "Parameters in CLR routines" on page 111
- "Routine invocation" on page 193
- "Parameter styles for external routines" on page 87
- "Scratchpads for UDFs and methods" on page 52
- "SQL in external routines" on page 101
- "Types of routines (procedures, functions, methods)" on page 5
- "Procedure result sets" on page 42

Related tasks:
- "Returning result sets from CLR procedures" on page 114
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running Applications*
- "Creating routines in the database" on page 31
- "Debugging routines" on page 38

Related reference:
- "Restrictions on CLR routines" on page 116

- "Supported SQL data types for the DB2 .NET Data Provider" on page 110

Related samples:

- "SpCreate.db2 -- Creates the external procedures implemented in spserver.cs"
- "SpServer.cs -- C# external code implementation of procedures created in spcat.db2"
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.vb"
- "SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2"

Supported SQL data types for the DB2 .NET Data Provider

The following table lists the mappings between the `DB2Type` data types in the DB2 .NET Data Provider, the DB2 data type, and the corresponding .NET Framework data type:

Table 4. Mapping DB2 Data Types to .NET data types

DB2Type Enum	DB2 Data Type	.NET Data Type
SmallInt	SMALLINT	Int16
Integer	INTEGER	Int32
BigInt	BIGINT	Int64
Real	REAL	Single
Double	DOUBLE PRECISION	Double
Float	FLOAT	Double
Decimal	DECIMAL	Decimal
Numeric	DECIMAL	Decimal
Date	DATE	DateTime
Time	TIME	TimeSpan
Timestamp	TIMESTAMP	DateTime
Char	CHAR	String
VarChar	VARCHAR	String
LongVarChar(1)	LONG VARCHAR	String
Binary	CHAR FOR BIT DATA	Byte[]
VarBinary	VARCHAR FOR BIT DATA	Byte[]
LongVarBinary(1)	LONG VARCHAR FOR BIT DATA	Byte[]
Graphic	GRAPHIC	String
VarGraphic	VARGRAPHIC	String
LongVarGraphic(1)	LONG GRAPHIC	String
Clob	CLOB	String
Blob	BLOB	Byte[]
DbClob	DBCLOB(N)	String

Notes:

1. These data types are not supported in DB2 .NET common language runtime routines. They are only supported in client applications.

Note: The `dbinfo` structure is passed into CLR functions and procedures as a parameter. The scratchpad and call type for CLR UDFs are also passed into CLR routines as parameters. For information about the appropriate CLR data types for these parameters, see the related topic:
- Parameters in CLR routines

Related concepts:
- "Parameter styles for external routines" on page 87
- "Common language runtime (CLR) routines" on page 106
- "Parameters in CLR routines" on page 111

Related tasks:
- "Passing structured type parameters to external routines" on page 292
- "Creating CLR routines" on page 107
- "Examples of CLR user-defined functions in C#" on page 139
- "Examples of CLR procedures in C#" on page 119

Related samples:
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.cs"
- "SpServer.cs -- C# external code implementation of procedures created in spcat.db2"
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.vb"
- "SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2"

Parameters in CLR routines

Parameter declaration in CLR routines must conform to the requirements of one of the supported parameter styles, and must respect the parameter keyword requirements of the particular .NET language used for the routine. If the routine is to use a scratchpad, the `dbinfo` structure, or to have a PROGRAM TYPE MAIN parameter interface, there are additional details to consider. This topic addresses all CLR parameter considerations.

Supported parameter styles for CLR routines:

The parameter style of the routine must be specified at routine creation time in the EXTERNAL clause of the CREATE statement for the routine. The parameter style must be accurately reflected in the implementation of the external CLR routine code. The following DB2® parameter styles are supported for CLR routines:
- `SQL` (Supported for procedures and functions)
- `GENERAL` (Supported for procedures only)
- `GENERAL WITH NULLS` (Supported for procedures only)
- `DB2SQL` (Supported for procedures and functions)

For more information about these parameter styles see:
- "Parameter styles for external routines" on page 87

CLR routine parameter null indicators:

If the parameter style chosen for a CLR routine requires that null indicators be specified for the parameters, the null indicators are to be passed into the CLR routine as System.Int16 type values, or in a System.Int16[] value when the parameter style calls for a vector of null indicators.

When the parameter style dictates that the null indicators be passed into the routine as distinct parameters, as is required for parameter style SQL, one System.Int16 null indicator is required for each parameter.

In .NET languages distinct parameters must be prefaced with a keywors to indicate if the parameter is passed by value or by reference. The same keyword that is used for a routine parameter must be used for the associated null indicator parameter. The keywords used to indicate whether an argument is passed by value or by reference are discussed in more detail below.

For more information about parameter style SQL and other supported parameter styles, see:

- "Parameter styles for external routines" on page 87

Passing CLR routine parameters by value or by reference:

.NET language routines that compile into intermediate language (IL) byte-code require that parameters be prefaced with keywords that indicate the particular properties of the parameter such as whether the parameter is passed by value, by reference, is an input only, or an output only parameter.

Parameter keywords are .NET language specific. For example to pass a parameter by reference in `C#`, the parameter keyword is `ref`, whereas in `Visual Basic`, a by reference parameter is indicated by the `byRef` keyword. The keywords must be used to indicate the SQL parameter usage (`IN`, `OUT`, `INOUT`) that was specified in the CREATE statement for the routine.

The following rules apply when applying parameter keywords to .NET language routine parameters in DB2 routines:

- `IN` type parameters must be declared *without* a parameter keyword in C#, and must be declared with the `byVal` keyword in `Visual Basic`.
- `INOUT` type parameters must be declared with the language specific keyword that indicates that the parameter is passed by reference. In `C#` the appropriate keyword is `ref`. In `Visual Basic`, the appropriate keyword is `byRef`.
- `OUT` type parameters must be declared with the language specific keyword that indicates that the parameter is an output only parameter. In `C#`, use the `out` keyword. In Visual Basic, the parameter must be declared with the `byRef` keyword. Output only parameters must always be assigned a value before the routine returns to the caller. If the routine does not assign a value to an output only parameter, an error will be raised when the .NET routine is compiled.

Here is what a C#, parameter style SQL procedure prototype looks like for a routine that returns a single output parameter `language`.

```
public static void Counter  (out String language,
                             out Int16  languageNullInd,
                             ref String sqlState,
                                 String funcName,
                                 String funcSpecName,
                             ref String sqlMsgString,
                             ref Byte[] scratchPad,
                                 Int32  callType);
```

It is clear that the parameter style SQL is implemented because of the extra null indicator parameter, `languageNullInd` associated with the output parameter `language`, the parameters for passing the SQLSTATE, the routine name, the routine specific name, and optional user-defined SQL error message. Parameter keywords have been specified for the parameters as follows:

- In C# no parameter keyword is required for input only parameters.
- In C# the 'out' keyword indicates that the variable is an output parameter only, and that its value has not been initialized by the caller.
- In C# the 'ref' keyword indicates that the parameter was initialized by the caller, and that the routine can optionally modify this value.

See the .NET language specific documentation regarding parameter passing to learn about the parameter keywords in that language.

Note:

DB2 controls allocation of memory for all parameters and maintains CLR references to all parameters passed into or out of a routine.

No parameter marker is required for procedure result sets:

No parameter markers is required in the procedure declaration of a procedure for a result set that will be returned to the caller. Any cursor statement that is not closed from inside of a CLR stored procedure will be passed back to its caller as a result set.

For more on result sets in CLR routines, see:

- "Returning result sets from CLR procedures" on page 114

Dbinfo structure as CLR parameter:

The `dbinfo` structure used for passing additional database information parameters to and from a routine is supported for CLR routines through the use of an IL `dbinfo` class. This class contains all of the elements found in the `C` language `sqludf_dbinfo` structure except for the length fields associated with the strings. The length of each string can be found using the .NET language `Length` property of the particular string.

To access the `dbinfo` class, simply include the `IBM®.Data.DB2` assembly in the file that contains your routine, and add a parameter of type `sqludf_dbinfo` to your routine's signature, in the position specified by the parameter style used.

UDF scratchpad as CLR parameter:

If a scratchpad is requested for a user defined function, it is passed into the routine as a `System.Byte[]` parameter of the specified size.

CLR UDF call type or final call parameter:

For user-defined functions that have requested a final call parameter or for table functions, the call type parameter is passed into the routine as a `System.Int32` data type.

PROGRAM TYPE MAIN supported for CLR procedures:

Program type MAIN is supported for .NET CLR procedures. Procedures defined as using Program Type MAIN must have the following signature:

```
void functionname(Int32 NumParams, Object[] Params)
```

Related concepts:
- "Parameter styles for external routines" on page 87
- "Procedure parameter modes" on page 42
- "Scratchpads for UDFs and methods" on page 52
- "Procedure result sets" on page 42
- "Parameter handling in PROGRAM TYPE MAIN or PROGRAM TYPE SUB procedures" on page 51

Related tasks:
- "Passing structured type parameters to external routines" on page 292
- "Examples of CLR user-defined functions in C#" on page 139
- "Examples of CLR procedures in C#" on page 119
- "Returning result sets from CLR procedures" on page 114

Related reference:
- "Supported SQL data types for the DB2 .NET Data Provider" on page 110

Returning result sets from CLR procedures

You can develop CLR procedures that return result sets to a calling routine or application. Result sets cannot be returned from CLR functions (UDFs).

The .NET representation of a result set is a `DB2DataReader` object which can be returned from one of the various execute calls of a `DB2Command` object. Any `DB2DataReader` object whose `Close()` method has not explicitly been called prior to the return of the procedure, can be returned. The order in which result sets are returned to the caller is the same as the order in which the `DB2DataReader` objects were instantiated. No additional parameters are required in the function definition in order to return a result set.

Prerequisites:

A general understanding of how to create CLR routines will help you to follow the steps in the procedure below returning results from a CLR procedure.

"Creating CLR routines" on page 107

Procedure:

To return a result set from a CLR procedure:
1. In the CREATE PROCEDURE statement for the CLR routine you must specify along with any other appropriate clauses, the DYNAMIC RESULT SETS clause with a value equal to the number of result sets that are to be returned by the procedure.
2. No parameter marker is required in the procedure declaration for a result set that is to be returned to the caller.

3. In the .NET language implementation of your CLR routine, create a `DB2Connection` object, a `DB2Command` object, and a `DB2Transaction` object. A `DB2Transaction` object is responsible for rolling back and committing database transactions.
4. Initialize the Transaction property of the `DB2Command` object to the `DB2Transaction` object.
5. Assign a string query to the `DB2Command` object's `CommandText` property that defines the result set that you want to return.
6. Instantiate a `DB2DataReader`, and assign to it, the result of the invocation of the `DB2Command` object method `ExecuteReader`. The result set of the query will be contained in the `DB2DataReader` object.
7. Do not execute the `Close()` method of the `DB2DataReader` object at any point prior to the procedure's return to the caller. The still open `DB2DataReader` object will be returned as a result set to the caller.

   When more than one `DB2DataReader` is left open upon the return of a procedure, the `DB2DataReaders` are returned to the caller in the order of their creation. Only the number of result sets specified in the CREATE PROCEDURE statement will be returned to the caller.
8. Compile your .NET CLR language procedure and install the assembly in the location specified by the EXTERNAL clause in the CREATE PROCEDURE statement. Execute the CREATE PROCEDURE statement for the CLR procedure, if you have not already done so.
9. Once the CLR procedure assembly has been installed in the appropriate location and the CREATE PROCEDURE statement has successfully been executed, you can invoke the procedure with the CALL statement to see the result sets return to the caller.

   For information on calling procedures and other types of routines:
   - "Routine invocation" on page 193

Specifying DYNAMIC RESULT SETS with a value greater than 1:

Only one dynamic result set can be returned from CLR procedures at this time. For details on this restriction see:
- "Restrictions on CLR routines" on page 116

Related concepts:
- "Routine invocation" on page 193
- "Procedure parameter modes" on page 42
- "Procedure result sets" on page 42
- "Common language runtime (CLR) routines" on page 106

Related tasks:
- "Creating CLR routines" on page 107

Related reference:
- "Restrictions on CLR routines" on page 116

Restrictions on CLR routines

The general implementation restrictions that apply to all external routines or particular routine classes (procedure or UDF) also apply to CLR routines. There are some restrictions that are particular to CLR routines. These restrictions are listed here.

The CREATE METHOD statement with LANGUAGE CLR clause is not supported:

You cannot create external methods for DB2 structured types that reference a CLR assembly. The use of a CREATE METHOD statement that specifies the LANGUAGE clause with value CLR is not supported.

CLR procedures cannot be implemented as NOT FENCED procedures:

CLR procedures cannot be run as unfenced procedures. The CREATE PROCEDURE statement for a CLR procedure can not specify the NOT FENCED clause.

At this time CLR procedures can return a maximum of one result set:

The maximum number of result sets that can be returned by a CLR procedure is limited to the maximum number of `DB2DataReader` objects that the data provider (`IBM.Data.DB2`) can simultaneously support having open within a connection. The maximum number that can be open at this time is one. Therefore only one result set can be returned from a CLR procedure.

If a CREATE PROCEDURE statement for a CLR procedure specifies the DYNAMIC RESULT SETS clause with a value greater than one, no error will be raised when this statement is executed.

At runtime however, only one `DB2DataReader` is allowed to be open in the procedure when the procedure returns. Therefore, only the single result set associated with the open `DB2DataReader` will be returned to the caller when the procedure returns.

Maximum decimal precision is 29, maximum decimal scale is 28 in a CLR routine:

The decimal data type in DB2 is represented with a precision of 31 digits and a scale of 28 digits. In .NET programming languages the decimal data type is represented with a precision of 29 digits and a scale of 28 digits. To avoid data truncation, DB2 external CLR routines must therefore not specify a decimal value that exceeds a precision of 29 digits and a scale of 28 digits.

If you require your routine to manipulate decimal values with the maximum precision and scale supported by DB2, you can implement your external routine in a different programming language such as C or Java.

Data types not supported in CLR routines:

The following DB2 SQL data types are not supported in CLR routines:
- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA

- LONG GRAPHIC
- DATALINK
- ROWID

Running a 32-bit CLR routine on a 64-bit instance:

CLR routines cannot be run on 64- bit instances, because the .NET Framework cannot be installed on 64-bit operating systems at this time.

Related concepts:
- "Common language runtime (CLR) routines" on page 106
- "Parameters in CLR routines" on page 111

Related tasks:
- "Creating CLR routines" on page 107
- "Returning result sets from CLR procedures" on page 114

Errors related to CLR routines

All external routines share a generally common implementation, there are some DB2 errors that may arise that are specific to CLR routines. This reference lists the most likely to be encountered of these errors, by their SQLCODE or behavior with some debugging suggestions. DB2 errors related to routines can be classified as follows:

Routine creation time errors
Errors that arise when the CREATE statement for the routine is executed.

Routine runtime errors
Errors that arise during the routine invocation or execution.

Regardless of when a DB2 routine related error is raised by DB2, the error message text details the cause of the error and the action that the user should take to resolve the problem. Additional routine error scenario information can be found in the `db2diag.log` diagnostic log file.

CLR routine creation time errors:

SQLCODE -451, SQLSTATE 42815
This error is raised upon an attempt to execute a CREATE TYPE statement that includes an external method declaration specifying the LANGUAGE clause with value CLR. You can not create DB2 external methods for structured types that reference a CLR assembly at this time. Change the LANGUAGE clause so that it specifies a supported language for the method and implement the method in that alternate language.

SQLCODE -449, SQLSTATE 42878
The CREATE statement for the CLR routine contains an invalidly formatted library or function identification in the EXTERNAL NAME clause. For language CLR, the EXTERNAL clause value must specifically take the form: `'<a>:<b>!<c>'` as follows:
- `<a>` is the CLR assembly file in which the class is located.
- `<b>` is the class in which the method to invoke resides.
- `<c>` is the method to invoke.

No leading or trailing blank characters are permitted between the single quotes, object identifiers, and the separating characters (for example, `' <a> ! <b> '` is invalid). Path and file names, however, may contain blanks if the platform permits. For all file names, the file can be specified using either the short form of the name (example: `math.dll` or the fully qualified path name (example: `d:\udfs\math.dll`. If the short form of the file name is used, if the platform is UNIX or if the routine is a LANGUAGE CLR routine, then the file must reside in the function directory. If the platform is Windows and the routine is not a LANGUAGE CLR routine then the file must reside in the system PATH. File extensions (examples: `.a` (on UNIX), `.dll` (on Windows)) should always be included in the file name.

CLR routine runtime errors:

SQLCODE -20282, SQLSTATE 42724, reason code 1
The external assembly specified by the EXTERNAL clause in the CREATE statement for the routine was not found.

- Check that the EXTERNAL clause specifies the correct routine assembly name and that the assembly is located in the specified location. If the EXTERNAL clause does not specify a fully qualified path name to the desired assembly, DB2 presumes that the path name provided is a relative path name to the assembly, relative to the DB2 function directory.

SQLCODE -20282, SQLSTATE 42724, reason code 2
An assembly was found in the location specified by the EXTERNAL clause in the CREATE statement for the routine, but no class was found within the assembly to match the class specified in the EXTERNAL clause.

- Check that the assembly name specified in the EXTERNAL clause is the correct assembly for the routine and that it exists in the specified location.
- Check that the class name specified in the EXTERNAL clause is the correct class name and that it exists in the specified assembly.

SQLCODE -20282, SQLSTATE 42724, reason code 3
An assembly was found in the location specified by the EXTERNAL clause in the CREATE statement for the routine, that had a correctly matching class definition, but the routine method signature does not match the routine signature specified in the CREATE statement for the routine.

- Check that the assembly name specified in the EXTERNAL clause is the correct assembly for the routine and that it exists in the specified location.
- Check that the class name specified in the EXTERNAL clause is the correct class name and that it exists in the specified assembly.
- Check that the parameter style implementation matches the parameter style specified in the CREATE statement for the routine.
- Check that the order of the parameter implementation matches the parameter declaration order in the CREATE statement for the routine and that it respects the extra parameter requirements for the parameter style.
- Check that the SQL parameter data types are correctly mapped to CLR .NET supported data types.

SQLCODE -4301, SQLSTATE 58004, reason code 5 or 6
An error occurred while attempting to start or communicate with a .NET

interpreter. DB2 was unable to load a dependent .NET library [reason code 5] or a call to the .NET interpreter failed [reason code 6].

- Ensure that the DB2 instance is configured correctly to run a .NET procedure or function (`mscoree.dll` must be present in the system `PATH`). Ensure that `db2clr.dll` is present in the `sqllib/bin` directory, and that `IBM.Data.DB2` is installed in the global assembly cache. If these are not present, please ensure that the `.NET Framework` version 1.1, or a later version, is installed on the database server, and that the database server is running DB2 version 8.2 or a later release.

SQLCODE -4302, SQLSTATE 38501
An unhandled exception occurred while executing, preparing to execute, or subsequent to executing the routine. This could be the result of a routine logic programming error that was unhandled or could be the result of an internal processing error.

Related concepts:
- "SQL in external routines" on page 101
- "Authorizations and binding of routines that contain SQL" on page 35
- "Security considerations for routines" on page 24
- "Library and class management considerations" on page 27
- "Common language runtime (CLR) routines" on page 106

Related tasks:
- "Creating CLR routines" on page 107

Examples of CLR procedures in C#

Once the basics of procedures, also called stored procedures, and the essentials of .NET common language runtime routines are understood, you can start using CLR procedures in your applications.

This topic contains examples of CLR procedures implemented in C# that illustrate the supported parameter styles, passing parameters, including the `dbinfo` structure, how to return a result set and more. For examples of CLR UDFs in C#:
- "Examples of CLR user-defined functions in C#" on page 139

Prerequisites:

Before working with the CLR procedure examples you may want to read the following concept topics:
- "Common language runtime (CLR) routines" on page 106
- "Creating CLR routines" on page 107
- "Routines in application development" on page 3
- Building common language runtime (CLR) .NET routines

The examples below make use of a table named `EMPLOYEE` that is contained in the `SAMPLE` database.

Procedure:

Use the following examples as references when making your own C# CLR procedures:

- “The C# external code file”
- “Example 1: C# parameter style GENERAL procedure”
- “Example 2: C# parameter style GENERAL WITH NULLS procedure” on page 121
- “Example 3: C# parameter style SQL procedure” on page 123
- “Example 4: C# procedure returning a result set” on page 126
- “Example 5: C# procedure accessing the dbinfo structure” on page 126
- “Example 6: C# procedure in PROGRAM TYPE MAIN style ” on page 127

The C# external code file:

The examples show a variety of C# procedure implementations. Each example consists of two parts: the CREATE PROCEDURE statement and the external C# code implementation of the procedure from which the associated assembly can be built.

The C# source file that contains the procedure implementations of the following examples is named `gwenProc.cs` and has the following format:

Table 5. C# external code file format

```
using System;
using System.IO;
using IBM.Data.DB2;

namespace bizLogic
{
   class empOps
   {         ...
     // C# procedures
             ...
   }
}
```

The file inclusions are indicated at the top of the file. The `IBM.Data.DB2` inclusion is required if any of the procedures in the file contain SQL. There is a namespace declaration in this file and a class `empOps` that contains the procedures. The use of namespaces is optional. If a namespace is used, the namespace must appear in the assembly path name provided in the EXTERNAL clause of the CREATE PROCEDURE statement.

It is important to note the name of the file, the namespace, and the name of the class, that contains a given procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that DB2 can locate the assembly and class of the CLR procedure.

Example 1: C# parameter style GENERAL procedure:

This example shows the following:
- CREATE PROCEDURE statement for a parameter style GENERAL procedure
- C# code for a parameter style GENERAL procedure

This procedure takes an employee ID and a current bonus amount as input. It retrieves the employee’s name and salary. If the current bonus amount is zero, a new bonus is calculated, based on the employee’s salary, and returned along with the employee’s full name. If the employee is not found, an empty string is returned.

Table 6. Code to create a C# parameter style GENERAL procedure

```
CREATE PROCEDURE setEmpBonusGEN(IN empID CHAR(6), INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
PROGRAM TYPE SUB
EXTERNAL NAME ’gwenProc.dll:bizLogic.empOps!SetEmpBonusGEN’ ;
```

```
 public static void SetEmpBonusGEN(    String empID,
                                   ref Decimal bonus,
                                   out String empName)
 {
    // Declare local variables
    Decimal salary = 0;

    DB2Command myCommand = DB2Context.GetCommand();
    myCommand.CommandText =
                     "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY "
                   + "FROM EMPLOYEE "
                   + "WHERE EMPNO = ’" + empID + ’";

    DB2DataReader reader = myCommand.ExecuteReader();

    if (reader.Read())  // If employee record is found
    {
       // Get the employee's full name and salary
       empName = reader.GetString(0) + " " +
                 reader.GetString(1) + ". " +
                 reader.GetString(2);

       salary = reader.GetDecimal(3);

       if (bonus == 0)
       {
          if (salary > 75000)
          {
             bonus = salary * (Decimal)0.025;
          }
          else
          {
             bonus = salary * (Decimal)0.05;
          }
       }
    }
    else  // Employee not found
    {
       empName = "";  // Set output parameter
    }

    reader.Close();
 }
```

Example 2: C# parameter style GENERAL WITH NULLS procedure:
This example shows the following:

- CREATE PROCEDURE statement for a parameter style GENERAL WITH NULLS procedure
- C# code for a parameter style GENERAL WITH NULLS procedure

This procedure takes an employee ID and a current bonus amount as input. If the input parameter is not null, it retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus based on salary is calculated and returned along with the employee's full name. If the employee data is not found, a NULL string and integer is returned.

Table 7. Code to create a C# parameter style GENERAL WITH NULLS procedure

```
CREATE PROCEDURE SetEmpbonusGENNULL(IN empID CHAR(6),
                                    INOUT bonus Decimal(9,2),
                                    OUT empName VARCHAR(60))
SPECIFIC SetEmpbonusGENNULL
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
;
```

Table 7. Code to create a C# parameter style GENERAL WITH NULLS procedure (continued)

```
public static void SetEmpBonusGENNULL(    String empID,
                                      ref Decimal bonus,
                                      out String empName,
                                          Int16[] NullInds)
{
   Decimal salary = 0;
   if (NullInds[0] == -1) // Check if the input is null
   {
     NullInds[1] = -1;    // Return a NULL bonus value
     empName = "";        // Set output value
     NullInds[2] = -1;    // Return a NULL empName value
   }
   else
   {
      DB2Command myCommand = DB2Context.GetCommand();
     myCommand.CommandText =
                       "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY "
                     + "FROM EMPLOYEE "
                     + "WHERE EMPNO = '" + empID + "'";
      DB2DataReader reader = myCommand.ExecuteReader();

      if (reader.Read())  // If employee record is found
      {
         // Get the employee's full name and salary
         empName = reader.GetString(0) + " "
         +
                   reader.GetString(1) + ". " +
                   reader.GetString(2);
         salary = reader.GetDecimal(3);

         if (bonus == 0)
         {
           if (salary > 75000)
           {
              bonus = salary * (Decimal)0.025;
              NullInds[1] = 0; // Return a non-NULL value
           }
           else
           {
              bonus = salary * (Decimal)0.05;
              NullInds[1] = 0; // Return a non-NULL value
           }
         }
      }
      else  // Employee not found
      {
         empName = "*sdq;;         // Set output parameter
         NullInds[2] = -1;     // Return a NULL value
      }

      reader.Close();
   }
}
```

Example 3: C# parameter style SQL procedure:
This example shows the following:

- CREATE PROCEDURE statement for a parameter style SQL procedure
- C# code for a parameter style SQL procedure

This procedure takes an employee ID and a current bonus amount as input. It retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus based on salary is calculated and returned along with the employee’s full name. If the employee is not found, an empty string is returned.

Table 8. Code to create a C# procedure in parameter style SQL with parameters

```
CREATE PROCEDURE SetEmpbonusSQL(IN empID CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpbonusSQL
LANGUAGE CLR
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME ’gwenProc.dll:bizLogic.empOps!SetEmpBonusSQL’ ;
```

Table 8. Code to create a C# procedure in parameter style SQL with parameters (continued)

```
public static void SetEmpBonusSQL(    String empID,
                                  ref Decimal bonus,
                                  out String empName,
                                      Int16  empIDNullInd,
                                  ref Int16  bonusNullInd,
                                  out Int16  empNameNullInd,
                                  ref string sqlStateate,
                                      string funcName,
                                      string specName,
                                  ref string sqlMessageText)
{
   // Declare local host variables
   Decimal salary eq; 0;

   if (empIDNullInd == -1) // Check if the input is null
   {
      bonusNullInd = -1;   // Return a NULL bonus value
      empName = "";
      empNameNullInd = -1; // Return a NULL empName value
   }
   else
      DB2Command myCommand = DB2Context.GetCommand();
      myCommand.CommandText =
                       "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY
                       "
                     + "FROM EMPLOYEE "
                     + "WHERE EMPNO = ’" + empID + "’";

      DB2DataReader reader = myCommand.ExecuteReader();

      if (reader.Read())  // If employee record is found
      {
         // Get the employee’s full name and salary
         empName = reader.GetString(0) + " "
         +
         reader.GetString(1) + ". " +
         reader.GetString(2);
         empNameNullInd = 0;
         salary = reader.GetDecimal(3);

         if (bonus == 0)
         {
            if (salary > 75000)
            {
               bonus = salary * (Decimal)0.025;
               bonusNullInd = 0;  // Return a non-NULL value
            }
            else
            {
               bonus = salary * (Decimal)0.05;
               bonusNullInd = 0;  // Return a non-NULL value
            }
         }
      }
      else  // Employee not found
      }
         empName = "";         // Set output parameter
         empNameNullInd = -1;  // Return a NULL value
      }

      reader.Close();
   }
}
```

Example 4: C# parameter style GENERAL procedure returning a result set: This example shows the following:

- CREATE PROCEDURE statement for an external C# procedure returning a result set
- C# code for a parameter style GENERAL procedure that returns a result set

This procedure accepts the name of a table as a parameter. It returns a result set containing all the rows of the table specified by the input parameter. This is done by leaving a `DB2DataReader` for a given query result set open when the procedure returns. Specifically, if `reader.Close()` is not executed, the result set will be returned.

Table 9. Code to create a C# procedure that returns a result set

```
CREATE PROCEDURE ReturnResultSet(IN tableName
VARCHAR(20))
SPECIFIC ReturnResultSet
DYNAMIC RESULT SETS 1
LANGUAGE CLR
PARAMETER STYLE GENERAL
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME
'gwenProc.dll:bizLogic.empOps!ReturnResultSet' ;
```

```
public static void ReturnResultSet(string tableName)
{
   DB2Command myCommand = DB2Context.GetCommand();

   // Set the SQL statement to be executed and execute it.
   myCommand.CommandText = "SELECT * FROM " + tableName;
   DB2DataReader reader = myCommand.ExecuteReader();

   // The DB2DataReader contains the result of the query.
   // This result set can be returned with the procedure,
   // by simply NOT closing the DB2DataReader.
   // Specifically, do NOT execute reader.Close();
}
```

Example 5: C# parameter style SQL procedure accessing the dbinfo structure: This example shows the following:

- CREATE PROCEDURE statement for a procedure accessing the `dbinfo` structure
- C# code for a parameter style SQL procedure that accesses the `dbinfo` structure

To access the `dbinfo` structure, the DBINFO clause must be specified in the CREATE PROCEDURE statement. No parameter is required for the `dbinfo` structure in the CREATE PROCEDURE statement however a parameter must be created for it, in the external routine code. This procedure returns only the value of the current database name from the `dbname` field in the `dbinfo` structure.

Table 10. Code to create a C# procedure that accesses the dbinfo structure

```
CREATE PROCEDURE ReturnDbName(OUT dbName VARCHAR(20))
SPECIFIC ReturnDbName
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE SQL
FENCED
DBINFO
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!ReturnDbName'
;
```

```
 public static void ReturnDbName(out string dbName,
                                 out Int16  dbNameNullInd,
                                 ref string sqlStateate,
                                     string funcName,
                                     string specName,
                                 ref string sqlMessageText,
                                     sqludf_dbinfo dbinfo)
 {
    // Retrieve the current database name from the
    // dbinfo structure and return it.
    // ** Note! ** dbinfo field names are case sensitive
    dbName = dbinfo.dbname;
    dbNameNullInd = 0;  // Return a non-null value;

    // If you want to return a user-defined error in
    // the SQLCA you can specify a 5 digit user-defined
    // sqlStateate and an error message string text.
    // For example:
    //
    //   sqlStateate = "ABCDE";
    //   sqlMessageText = "A user-defined error has occured"
    //
    //  DB2 returns the above values to the client in the
    //  SQLCA structure.  The values are used to generate a
    //  standard DB2 sqlStateate error.
 }
```

Example 6: C# procedure with PROGRAM TYPE MAIN style:
This example shows the following:

- CREATE PROCEDURE statement for a procedure using a main program style
- C# parameter style GENERAL WITH NULLS code in using a MAIN program style

To implement a routine in a main program style, the PROGRAM TYPE clause must be specified in the CREATE PROCEDURE statement with the value MAIN. Parameters are specified in the CREATE PROCEDURE statement however in the code implementation, parameters are passed into the routine in an `argc` integer parameter and an `argv` array of parameters.

Table 11. Code to create a C# procedure in program type MAIN style

```
CREATE PROCEDURE MainStyle( IN empID CHAR(6),
                            INOUT bonus Decimal(9,2),
                            OUT empName VARCHAR(60))
SPECIFIC MainStyle
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
FENCED
PROGRAM TYPE MAIN
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!main' ;
```

Table 11. Code to create a C# procedure in program type MAIN style (continued)

```
public static void main(Int32 argc, Object[]
argv)
{
  String empID = (String)argv[0];   // argv[0] has nullInd:argv[3]
  Decimal bonus = (Decimal)argv[1]; // argv[1] has nullInd:argv[4]
                                    // argv[2] has nullInd:argv[5]
  Decimal salary = 0;     Int16[] NullInds =
  (Int16[])argv[3];

  if ((NullInds[0]) == (Int16)(-1)) // Check if empID is null
  {
     NullInds[1] = (Int16)(-1);  // Return a NULL bonus value
     argv[1] = (String)"";       // Set output parameter empName
     NullInds[2] = (Int16)(-1);  // Return a NULL empName value
     Return;
  }
  else
     DB2Command myCommand = DB2Context.GetCommand();
     myCommand.CommandText =
                      "SELECT FIRSTNME, MIDINIT, LASTNAME, salary "
                    + "FROM EMPLOYEE "
                    + "WHERE EMPNO = '" + empID + "'";

     DB2DataReader reader = myCommand.ExecuteReader();

     if (reader.Read())  // If employee record is found
     {
       // Get the employee's full name and salary
       argv[2] = (String) (reader.GetString(0) + " " +
                           reader.GetString(1) + ".
                           " +
                           reader.GetString(2));
       NullInds[2] = (Int16)0;
       salary = reader.GetDecimal(3);

       if (bonus == 0)
       {
          if (salary > 75000)
          {
             argv[1] = (Decimal)(salary * (Decimal)0.025);
             NullInds[1] = (Int16)(0);  // Return a non-NULL value
          }
          else
          {
             argv[1] = (Decimal)(salary * (Decimal)0.05);
             NullInds[1] = (Int16)(0);  // Return a non-NULL value
          }
       }
     }
     else  // Employee not found
     {
        argv[2] = (String)("");         // Set output parameter
        NullInds[2] = (Int16)(-1);      // Return a NULL value
     }

     reader.Close();
   }
}
```

Related concepts:

- "Common language runtime (CLR) routines" on page 106
- "Routines in application development" on page 3

Related tasks:
- "Examples of CLR user-defined functions in C#" on page 139
- "Creating CLR routines" on page 107
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running Applications*

Related samples:
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.cs"
- "SpServer.cs -- C# external code implementation of procedures created in spcat.db2"
- "SpCreate.db2 -- Creates the external procedures implemented in spserver.vb"
- "SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2"

Examples of CLR procedures in Visual Basic

Once the basics of procedures, also called stored procedures, and the essentials of .NET common language runtime routines are understood, you can start using CLR procedures in your applications.

This topic contains examples of CLR procedures implemented in `Visual Basic`; that illustrate the supported parameter styles, passing parameters, including the `dbinfo` structure, how to return a result set and more. For examples of CLR UDFs in `Visual Basic`:
- "Examples of CLR user-defined functions in Visual Basic" on page 145

Prerequisites:

Before working with the CLR procedure examples you may want to read the following concept topics:
- "Common language runtime (CLR) routines" on page 106
- "Creating CLR routines" on page 107
- "Routines in application development" on page 3
- Building common language runtime (CLR) .NET routines

The examples below make use of a table named `EMPLOYEE` that is contained in the `SAMPLE` database.

Procedure:

Use the following examples as references when making your own Visual Basic CLR procedures:
- "The Visual Basic external code file"
- "Example 1: Visual Basic parameter style GENERAL procedure" on page 131
- "Example 2: Visual Basic parameter style GENERAL WITH NULLS procedure" on page 132
- "Example 3: Visual Basic parameter style SQL procedure" on page 134
- "Example 4: Visual Basic procedure returning a result set" on page 135
- "Example 5: Visual Basic procedure accessing the dbinfo structure" on page 136
- "Example 6: Visual Basic procedure in PROGRAM TYPE MAIN style " on page 137

The Visual Basic external code file:

The examples show a variety of Visual Basic procedure implementations. Each example consists of two parts: the CREATE PROCEDURE statement and the external Visual Basic code implementation of the procedure from which the associated assembly can be built.

The Visual Basic source file that contains the procedure implementations of the following examples is named `gwenVbProc.vb` and has the following format:

Table 12. Visual Basic external code file format

```
using System;
using System.IO;
using IBM.Data.DB2;

Namespace bizLogic

   Class empOps
             ...
     ' Visual Basic procedures
             ...
   End Class
End Namespace
```

The file inclusions are indicated at the top of the file. The `IBM.Data.DB2` inclusion is required if any of the procedures in the file contain SQL. There is a namespace declaration in this file and a class `empOps` that contains the procedures. The use of namespaces is optional. If a namespace is used, the namespace must appear in the assembly path name provided in the EXTERNAL clause of the CREATE PROCEDURE statement.

It is important to note the name of the file, the namespace, and the name of the class, that contains a given procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that DB2 can locate the assembly and class of the CLR procedure.

Example 1: Visual Basic parameter style GENERAL procedure:

This example shows the following:
- CREATE PROCEDURE statement for a parameter style GENERAL procedure
- Visual Basic code for a parameter style GENERAL procedure

This procedure takes an employee ID and a current bonus amount as input. It retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus is calculated, based on the employee salary, and returned along with the employee's full name. If the employee is not found, an empty string is returned.

Table 13. Code to create a Visual Basic parameter style GENERAL procedure

```
CREATE PROCEDURE SetEmpBonusGEN(IN empId CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC setEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusGEN'
```

```
  Public Shared Sub SetEmpBonusGEN(ByVal empId As String, _
                                   ByRef bonus As Decimal, _
                                   ByRef empName As String)

     Dim salary As Decimal
     Dim myCommand As DB2Command
     Dim myReader As DB2DataReader

     salary = 0

     myCommand = DB2Context.GetCommand()
     myCommand.CommandText = _
                "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
              + "FROM EMPLOYEE " _
              + "WHERE EMPNO = '" + empId + "'"
     myReader = myCommand.ExecuteReader()

     If myReader.Read()  ' If employee record is found
        ' Get the employee's full name and salary
        empName = myReader.GetString(0) + " " _
                + myReader.GetString(1) + ". " _
                + myReader.GetString(2)

        salary = myReader.GetDecimal(3)

        If bonus = 0
           If salary > 75000
              bonus = salary * 0.025
           Else
              bonus = salary * 0.05
           End If
        End If
     Else  ' Employee not found
        empName = ""  ' Set output parameter
     End If

     myReader.Close()

  End Sub
```

Example 2: Visual Basic parameter style GENERAL WITH NULLS procedure: This example shows the following:

- CREATE PROCEDURE statement for a parameter style GENERAL WITH NULLS procedure
- Visual Basic code for a parameter style GENERAL WITH NULLS procedure

This procedure takes an employee ID and a current bonus amount as input. If the input parameter is not null, it retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus based on salary is calculated and returned along with the employee's full name. If the employee data is not found, a NULL string and integer is returned.

Table 14. Code to create a Visual Basic parameter style GENERAL WITH NULLS procedure

```
CREATE PROCEDURE SetEmpBonusGENNULL(IN empId CHAR(6),
                                    INOUT bonus Decimal(9,2),
                                    OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusGENNULL
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
```

```
  Public Shared Sub SetEmpBonusGENNULL(ByVal empId As String, _
                                       ByRef bonus As Decimal, _
                                       ByRef empName As String, _
                                       byVal nullInds As Int16())

     Dim salary As Decimal
     Dim myCommand As DB2Command
     Dim myReader As DB2DataReader

     salary = 0

     If nullInds(0) = -1   ' Check if the input is null
        nullInds(1) = -1   ' Return a NULL bonus value
        empName = ""       ' Set output parameter
        nullInds(2) = -1   ' Return a NULL empName value
        Return
     Else
        myCommand = DB2Context.GetCommand()
        myCommand.CommandText = _
                   "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
                 + "FROM EMPLOYEE " _
                 + "WHERE EMPNO = '" + empId + "'"

        myReader = myCommand.ExecuteReader()

        If myReader.Read()  ' If employee record is found
           ' Get the employee's full name and salary
           empName = myReader.GetString(0) + " " _
                   + myReader.GetString(1) + ". " _
                   + myReader.GetString(2)

           salary = myReader.GetDecimal(3)

           If bonus = 0
              If salary > 75000
                 bonus = Salary * 0.025
                 nullInds(1) = 0 'Return a non-NULL value
              Else
                 bonus = salary * 0.05
                 nullInds(1) = 0 ' Return a non-NULL value
              End If
           Else  'Employee not found
              empName = ""       ' Set output parameter
              nullInds(2) = -1   ' Return a NULL value
           End If
        End If

        myReader.Close()

     End If

  End Sub
```

Example 3: Visual Basic parameter style SQL procedure:
This example shows the following:

- CREATE PROCEDURE statement for a parameter style SQL procedure
- Visual Basic code for a parameter style SQL procedure

This procedure takes an employee ID and a current bonus amount as input. It retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus based on salary is calculated and returned along with the employee's full name. If the employee is not found, an empty string is returned.

Table 15. Code to create a Visual Basic procedure in parameter style SQL with parameters

```
CREATE PROCEDURE SetEmpBonusSQL(IN empId CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusSQL
LANGUAGE CLR
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusSQL'
```

Table 15. Code to create a Visual Basic procedure in parameter style SQL with parameters (continued)

```
   Public Shared Sub SetEmpBonusSQL(byVal empId As String, _
                                    byRef bonus As Decimal, _
                                    byRef empName As String, _
                                    byVal empIdNullInd As Int16, _
                                    byRef bonusNullInd As Int16, _
                                    byRef empNameNullInd As Int16, _
                                    byRef sqlState As String, _
                                    byVal funcName As String, _
                                    byVal specName As String, _
                                    byRef sqlMessageText As String)

      ' Declare local host variables
      Dim salary As Decimal
      Dim myCommand As DB2Command
      Dim myReader As DB2DataReader

      salary = 0

      If empIdNullInd = -1   ' Check if the input is null
         bonusNullInd = -1   ' Return a NULL Bonus value
         empName = ""
         empNameNullInd = -1 ' Return a NULL empName value
      Else
        myCommand = DB2Context.GetCommand()
        myCommand.CommandText = _
                   "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
                 + "FROM EMPLOYEE " _
                 + " WHERE EMPNO = '" + empId + "'"

         myReader = myCommand.ExecuteReader()

         If myReader.Read()  ' If employee record is found
            '  Get the employee's full name and salary
            empName = myReader.GetString(0) + " "
                    + myReader.GetString(1) _
                    + ". " +  myReader.GetString(2)
            empNameNullInd = 0
            salary = myReader.GetDecimal(3)

            If bonus = 0
               If salary > 75000
                  bonus = salary * 0.025
                  bonusNullInd = 0  ' Return a non-NULL value
               Else
                  bonus = salary * 0.05
                  bonusNullInd = 0  ' Return a non-NULL value
               End If
            End If
         Else  ' Employee not found
            empName = ""            ' Set output parameter
            empNameNullInd = -1     ' Return a NULL value
         End If

         myReader.Close()
      End If

   End Sub
```

Example 4: Visual Basic parameter style GENERAL procedure returning a result set:
This example shows the following:

- CREATE PROCEDURE statement for an external Visual Basic procedure returning a result set
- Visual Basic code for a parameter style GENERAL procedure that returns a result set

This procedure accepts the name of a table as a parameter. It returns a result set containing all the rows of the table specified by the input parameter. This is done by leaving a `DB2DataReader` for a given query result set open when the procedure returns. Specifically, if `reader.Close()` is not executed, the result set will be returned.

Table 16. Code to create a Visual Basic procedure that returns a result set

```
CREATE PROCEDURE ReturnResultSet(IN tableName VARCHAR(20))
SPECIFIC ReturnResultSet
DYNAMIC RESULT SETS 1
LANGUAGE CLR
PARAMETER STYLE GENERAL
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!ReturnResultSet'
```

```
Public Shared Sub ReturnResultSet(byVal tableName As String)

      Dim myCommand As DB2Command
      Dim myReader As DB2DataReader

      myCommand = DB2Context.GetCommand()

      ' Set the SQL statement to be executed and execute it.
      myCommand.CommandText = "SELECT * FROM " + tableName
      myReader = myCommand.ExecuteReader()

      ' The DB2DataReader contains the result of the query.
      ' This result set can be returned with the procedure,
      ' by simply NOT closing the DB2DataReader.
      ' Specifically, do NOT execute reader.Close()

    End Sub
```

Example 5: Visual Basic parameter style SQL procedure accessing the dbinfo structure:
This example shows the following:

- CREATE PROCEDURE statement for a procedure accessing the `dbinfo` structure
- Visual Basic code for a parameter style SQL procedure that accesses the `dbinfo` structure

To access the `dbinfo` structure, the DBINFO clause must be specified in the CREATE PROCEDURE statement. No parameter is required for the `dbinfo` structure in the CREATE PROCEDURE statement however a parameter must be created for it, in the external routine code. This procedure returns only the value of the current database name from the `dbname` field in the `dbinfo` structure.

Table 17. Code to create a Visual Basic procedure that accesses the dbinfo structure

```
CREATE PROCEDURE ReturnDbName(OUT dbName VARCHAR(20))
SPECIFIC ReturnDbName
LANGUAGE CLR
PARAMETER STYLE SQL
DBINFO
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!ReturnDbName'
```

```
  Public Shared Sub ReturnDbName(byRef dbName As String, _
                                 byRef dbNameNullInd As Int16, _
                                 byRef sqlState As String, _
                                 byVal funcName As String, _
                                 byVal specName As String, _
                                 byRef sqlMessageText As String, _
                                 byVal dbinfo As sqludf_dbinfo)

    ' Retrieve the current database name from the
    ' dbinfo structure and return it.
    dbName = dbinfo.dbname
    dbNameNullInd = 0  ' Return a non-null value

    ' If you want to return a user-defined error in
    ' the SQLCA you can specify a 5 digit user-defined
    ' SQLSTATE and an error message string text.
    ' For example:
    '
    ' sqlState = "ABCDE"
    ' msg_token = "A user-defined error has occured"
    '
    ' These will be returned by DB2 in the SQLCA.  It
    ' will appear in the format of a regular DB2 sqlState
    ' error.
  End Sub
```

Example 6: Visual Basic procedure with PROGRAM TYPE MAIN style:
This example shows the following:

- CREATE PROCEDURE statement for a procedure using a main program style
- Visual Basic parameter style GENERAL WITH NULLS code in using a MAIN program style

To implement a routine in a main program style, the PROGRAM TYPE clause must be specified in the CREATE PROCEDURE statement with the value MAIN. Parameters are specified in the CREATE PROCEDURE statement however in the code implementation, parameters are passed into the routine in an `argc` integer parameter and an `argv` array of parameters.

Table 18. Code to create a Visual Basic procedure in program type MAIN style

```
CREATE PROCEDURE MainStyle(IN empId CHAR(6),
                           INOUT bonus Decimal(9,2),
                           OUT empName VARCHAR(60))
SPECIFIC mainStyle
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
FENCED
PROGRAM TYPE MAIN
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!Main'
```

Table 18. Code to create a Visual Basic procedure in program type MAIN style (continued)

```
   Public Shared Sub Main( byVal argc As Int32, _
                           byVal argv As Object())

      Dim myCommand As DB2Command
      Dim myReader As DB2DataReader
      Dim empId As String
      Dim bonus As Decimal
      Dim salary As Decimal
      Dim nullInds As Int16()

      empId = argv(0)  ' argv[0] (IN)    nullInd = argv[3]
      bonus = argv(1)  ' argv[1] (INOUT) nullInd = argv[4]
                       ' argv[2] (OUT)   nullInd = argv[5]
      salary = 0
      nullInds = argv(3)

      If nullInds(0) = -1     ' Check if the empId input is null
         nullInds(1) = -1     ' Return a NULL Bonus value
         argv(1) = ""         ' Set output parameter empName
         nullInds(2) = -1     ' Return a NULL empName value
         Return
      Else
         ' If the employee exists and the current bonus is 0,
         ' calculate a new employee bonus based on the employee's
         ' salary.  Return the employee name and the new bonus
         myCommand = DB2Context.GetCommand()
         myCommand.CommandText = _
                   "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
                 + " FROM EMPLOYEE " _
                 + " WHERE EMPNO = '" + empId + "'"

         myReader = myCommand.ExecuteReader()


         If myReader.Read()  ' If employee record is found
            ' Get the employee's full name and salary
            argv(2) = myReader.GetString(0) + " " _
                    + myReader.GetString(1) + ". " _
                    + myReader.GetString(2)
            nullInds(2) = 0
            salary = myReader.GetDecimal(3)

            If bonus = 0
               If salary > 75000
                  argv(1) = salary * 0.025
                  nullInds(1) = 0  ' Return a non-NULL value
               Else
                  argv(1) = Salary * 0.05
                  nullInds(1) = 0  ' Return a non-NULL value
               End If
            End If
         Else  ' Employee not found
            argv(2) = ""      ' Set output parameter
            nullInds(2) = -1  ' Return a NULL value
         End If

         myReader.Close()
      End If

   End Sub
```

Related concepts:
- "Common language runtime (CLR) routines" on page 106
- "Routines in application development" on page 3

Related tasks:

- "Examples of CLR user-defined functions in Visual Basic" on page 145
- "Creating CLR routines" on page 107
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running Applications*

Examples of CLR user-defined functions in C#

Once you understand the basics of user-defined functions (UDFs), and the essentials of CLR routines, you can start exploiting CLR UDFs in your applications and database environment. This topic contains some examples of CLR UDFs to get you started. For examples of CLR procedures in C#:

- "Examples of CLR procedures in C#" on page 119

Prerequisites:

Before working with the CLR UDF examples you may want to read the following concept topics:

- "Common language runtime (CLR) routines" on page 106
- "Creating CLR routines" on page 107
- "User-defined scalar functions" on page 13
- "User-defined scalar functions" on page 15
- Building common language runtime (CLR) .NET routines

The examples below make use of a table named `EMPLOYEE` that is contained in the `SAMPLE` database.

Procedure:

Use the following examples as references when making your own C# CLR UDFs:

- "The C# external code file"
- "Example 1: C# parameter style SQL table function" on page 140
- "Example 2: C# parameter style SQL scalar function" on page 143

The C# external code file:

The following examples show a variety of C# UDF implementations. The CREATE FUNCTION statement is provided for each UDF with the corresponding C# source code from which the associated assembly can be built. The C# source file that contains the functions declarations used in the following examples is named `gwenUDF.cs` and has the following format:

Table 19. C# external code file format

```
using System;
using System.IO;
using IBM.Data.DB2;

namespace bizLogic
{
   ...
   // Class definitions that contain UDF declarations
   // and any supporting class definitions
   ...
}
```

The function declarations must be contained in a class within a C# file. The use of namespaces is optional. If a namespace is used, the namespace must appear in the assembly path name provided in the EXTERNAL clause of the CREATE PROCEDURE statement. The `IBM.Data.DB2.` inclusion is required if the function contains SQL.

Example 1: C# parameter style SQL table function:

This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL table function
- C# code for a parameter style SQL table function

This table function returns a table containing rows of employee data that was created from a data array. There are two classes associated with this example. Class `person` represents the employees, and the class `empOps` contains the routine table UDF that uses class `person`. The employee salary information is updated based on the value of an input parameter. The data array in this example is created within the table function itself on the first call of the table function. Such an array could have also been created by reading in data from a text file on the filesystem. The array data values are written to a scratchpad so that the data can be accessed in subsequent calls of the table function.

On each call of the table function, one record is read from the array and one row is generated in the table that is returned by the function. The row is generated in the table, by setting the output parameters of the table function to the desired row values. After the final call of the table function occurs, the table of generated rows is returned.

Table 20. Code to create a C# parameter style SQL table function

```
CREATE FUNCTION tableUDF(double)
RETURNS TABLE (name varchar(20),
               job varchar(20),
               salary double)
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!tableUDF'
LANGUAGE CLR
PARAMETER STYLE SQL
NOT DETERMINISTIC
FENCED
SCRATCHPAD 10
FINAL CALL
DISALLOW PARALLEL
NO DBINFO
```

Table 20. Code to create a C# parameter style SQL table function (continued)

```
// The class Person is a supporting class for
// the table function UDF, tableUDF, below.
class Person
{
     private String name;
     private String position;
     private Int32 salary;

     public Person(String newName, String newPosition, Int32
     newSalary)
     {
        this.name = newName;
        this.position = newPosition;
        this.salary = newSalary;
     }

     public String getName()
     {
        return this.name;
     }

     public String getPosition()
     {
        return this.position;
     }

     public Int32 getSalary()
     {
        return this.salary;
     }
 }
```

Table 20. Code to create a C# parameter style SQL table function (continued)

```
class empOps
{
  {
   public static void TableUDF( Double factor, out String name,
                    out String position, out Double salary,
                    Int16 factorNullInd, out Int16 nameNullInd,
                    out Int16 positionNullInd, out Int16 salaryNullInd,
                    ref String sqlState, String funcName,
                    String specName, ref String sqlMessageText,
                    Byte[] scratchPad, Int32 callType)
   {

      Int16 intRow = 0;

      // Create an array of Person type information
      Person[] Staff = new
      Person[3];
      Staff[0] = new Person("Gwen", "Developer", 10000);
      Staff[1] = new Person("Andrew", "Developer", 20000);
      Staff[2] = new Person("Liu", "Team Leader", 30000);

      salary = 0;
      name = position = "";
      nameNullInd = positionNullInd = salaryNullInd = -1;

      switch(callType)
      {
         case (-2):  // Case SQLUDF_TF_FIRST:
           break;

         case (-1):  // Case SQLUDF_TF_OPEN:
           intRow = 1;
           scratchPad[0] = (Byte)intRow;  // Write to scratchpad
           break;
         case (0):   // Case SQLUDF_TF_FETCH:
           intRow = (Int16)scratchPad[0];
           if (intRow > Staff.Length)
           {
              sqlState = "02000";  // Return an error SQLSTATE
           }
           else
           {
              // Generate a row in the output table
              // based on the Staff array data.
              name =
              Staff[intRow-1].getName();
              position = Staff[intRow-1].getPosition();
              salary = (Staff[intRow-1].getSalary[]] * factor;
              nameNullInd = 0;
              positionNullInd = 0;
              salaryNullInd = 0;
           }
           intRow++;
           scratchPad[0] = (Byte)intRow;  // Write scratchpad
           break;

         case (1):   // Case SQLUDF_TF_CLOSE:
           break;

         case (2):   // Case SQLUDF_TF_FINAL:
           break;
      }
   }
}
```

Example 2: C# parameter style SQL scalar function:
This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL scalar function
- C# code for a parameter style SQL scalar function

This scalar function returns a single count value for each input value that it operates on. For an input value in the nth position of the set of input values, the output scalar value is the value n. On each call of the scalar function, where one call is associated with each row or value in the input set of rows or values, the count is increased by one and the current value of the count is returned. The count is then saved in the scratchpad memory buffer to maintain the count value between each call of the scalar function.

This scalar function can be easily invoked if for example we have a table defined as follows:

```
CREATE TABLE T (i1 INTEGER);
INSERT INTO T VALUES 12, 45, 16, 99;
```

A simple query such as the following can be used to invoke the scalar function:

```
SELECT countUp(i1) as count, i1 FROM T;
```

The output of such a query would be:

```
COUNT           I1
-----------     ----------
1               12
2               45
3               16
4               99
```

This scalar UDF is quite simple. Instead of returning just the count of the rows, you could use a scalar function to format data in an existing column. For example you might append a string to each value in an address column or you might build up a complex string from a series of input strings or you might do a complex mathematical evaluation over a set of data where you must store an intermediate result.

Table 21. Code to create a C# parameter style SQL scalar function

```
CREATE FUNCTION countUp(INTEGER)
RETURNS INTEGER
LANGUAGE CLR
PARAMETER STYLE SQL
FENCED
SCRATCHPAD 10
FINAL CALL
VARIANT
NO SQL
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp' ;
```

Table 21. Code to create a C# parameter style SQL scalar function (continued)

```
class empOps
{
   public static void CountUp(     Int32 input,
                               out Int32 outCounter,
                                   Int16 inputNullInd,
                               out Int16 outCounterNullInd,
                               ref String sqlState,
                                   String funcName,
                                   String specName,
                               ref String sqlMessageText,
                                   Byte[] scratchPad,
                                   Int32 callType)
   {
      Int32 counter = 1;        switch(callType)
      {
         case -1: // case SQLUDF_FIRST_CALL
           scratchPad[0] = (Byte)counter;
           outCounter = counter;
           outCounterNullInd = 0;
           break;
         case 0:  // case SQLUDF_NORMAL_CALL:
           counter = (Int32)scratchPad[0];
           counter = counter + 1;
           outCounter = counter;
           outCounterNullInd = 0;
           scratchPad[0] =
           (Byte)counter;
           break;
         case 1:  // case SQLUDF_FINAL_CALL:
           counter =
           (Int32)scratchPad[0];
           outCounter = counter;
           outCounterNullInd = 0;
           break;
         default: // Should never enter here
                  // * Required so that at compile time
                  //   out parameter outCounter is always set *
           outCounter = (Int32)(0);
           outCounterNullInd = -1;
           sqlState="ABCDE";
           sqlMessageText = "Should not get here: Default
           case!";
           break;
      }
   }
}
```

Related concepts:

- "Common language runtime (CLR) routines" on page 106
- "User-defined scalar functions" on page 13
- "User-defined scalar functions" on page 15

Related tasks:

- "Examples of CLR procedures in C#" on page 119
- "Creating CLR routines" on page 107
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running Applications*

Related samples:

- “SpCreate.db2 -- Creates the external procedures implemented in spserver.cs”
- “SpServer.cs -- C# external code implementation of procedures created in spcat.db2”
- “SpCreate.db2 -- Creates the external procedures implemented in spserver.vb”
- “SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2”

Examples of CLR user-defined functions in Visual Basic

Once you understand the basics of user-defined functions (UDFs), and the essentials of CLR routines, you can start exploiting CLR UDFs in your applications and database environment. This topic contains some examples of CLR UDFs to get you started. For examples of CLR procedures in Visual Basic:
- “Examples of CLR procedures in Visual Basic” on page 130

Prerequisites:

Before working with the CLR UDF examples you may want to read the following concept topics:
- “Common language runtime (CLR) routines” on page 106
- “Creating CLR routines” on page 107
- “User-defined scalar functions” on page 13
- “User-defined scalar functions” on page 15
- Building common language runtime (CLR) .NET routines

The examples below make use of a table named `EMPLOYEE` that is contained in the `SAMPLE` database.

Procedure:

Use the following examples as references when making your own Visual Basic CLR UDFs:
- “The Visual Basic external code file”
- “Example 1: Visual Basic parameter style SQL table function” on page 146
- “Example 2: Visual Basic parameter style SQL scalar function” on page 148

The Visual Basic external code file:

The following examples show a variety of Visual Basic UDF implementations. The CREATE FUNCTION statement is provided for each UDF with the corresponding Visual Basic source code from which the associated assembly can be built. The Visual Basic source file that contains the functions declarations used in the following examples is named `gwenVbUDF.cs` and has the following format:

Table 22. Visual Basic external code file format

```
 using System;
 using System.IO;
 using IBM.Data.DB2;

 Namespace bizLogic

    ...
    ' Class definitions that contain UDF declarations
    ' and any supporting class definitions
    ...

End Namespace
```

The function declarations must be contained in a class within a Visual Basic file. The use of namespaces is optional. If a namespace is used, the namespace must appear in the assembly path name provided in the EXTERNAL clause of the CREATE PROCEDURE statement. The `IBM.Data.DB2.` inclusion is required if the function contains SQL.

Example 1: Visual Basic parameter style SQL table function:
This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL table function
- Visual Basic code for a parameter style SQL table function

This table function returns a table containing rows of employee data that was created from a data array. There are two classes associated with this example. Class `person` represents the employees, and the class `empOps` contains the routine table UDF that uses class `person`. The employee salary information is updated based on the value of an input parameter. The data array in this example is created within the table function itself on the first call of the table function. Such an array could have also been created by reading in data from a text file on the filesystem. The array data values are written to a scratchpad so that the data can be accessed in subsequent calls of the table function.

On each call of the table function, one record is read from the array and one row is generated in the table that is returned by the function. The row is generated in the table, by setting the output parameters of the table function to the desired row values. After the final call of the table function occurs, the table of generated rows is returned.

Table 23. Code to create a Visual Basic parameter style SQL table function

```
 CREATE FUNCTION TableUDF(double)
 RETURNS TABLE (name varchar(20),
               job varchar(20),
               salary double)
 EXTERNAL NAME 'gwenVbUDF.dll:bizLogic.empOps!TableUDF'
 LANGUAGE CLR
 PARAMETER STYLE SQL
 NOT DETERMINISTIC
 FENCED
 SCRATCHPAD 10
 FINAL CALL
 DISALLOW PARALLEL
 NO DBINFO
```

Table 23. Code to create a Visual Basic parameter style SQL table function (continued)

```
Class Person
' The class Person is a supporting class for
' the table function UDF, tableUDF, below.

  Private name As String
  Private position As String
  Private salary As Int32

  Public Sub New(ByVal newName As String, _
                 ByVal newPosition As String, _
                 ByVal newSalary As Int32)

    name = newName
    position = newPosition
    salary = newSalary
  End Sub

  Public Property GetName() As String
    Get
      Return name
    End Get

    Set (ByVal value As String)
      name = value
    End Set
  End Property

  Public Property GetPosition() As String
    Get
      Return position
    End Get

    Set (ByVal value As String)
      position = value
    End Set
  End Property

  Public Property GetSalary() As Int32
    Get
      Return salary
    End Get

    Set (ByVal value As Int32)
      salary = value
    End Set
  End Property

End Class
```

Table 23. Code to create a Visual Basic parameter style SQL table function (continued)

```
Class empOps

  Public Shared Sub TableUDF(byVal factor as Double, _
                             byRef name As String, _
                             byRef position As String, _
                             byRef salary As Double, _
                             byVal factorNullInd As Int16, _
                             byRef nameNullInd As Int16, _
                             byRef positionNullInd As Int16, _
                             byRef salaryNullInd As Int16, _
                             byRef sqlState As String, _
                             byVal funcName As String, _
                             byVal specName As String, _
                             byRef sqlMessageText As String, _
                             byVal scratchPad As Byte(), _
                             byVal callType As Int32)

    Dim intRow As Int16

    intRow = 0

    ' Create an array of Person type information
    Dim staff(2) As Person
    staff(0) = New Person("Gwen", "Developer", 10000)
    staff(1) = New Person("Andrew", "Developer", 20000)
    staff(2) = New Person("Liu", "Team Leader", 30000)

    ' Initialize output parameter values and NULL indicators
    salary = 0
    name = position = ""
    nameNullInd = positionNullInd = salaryNullInd = -1

    Select callType
       Case -2   ' Case SQLUDF_TF_FIRST:
       Case -1   ' Case SQLUDF_TF_OPEN:
         intRow = 1
         scratchPad(0) = intRow  ' Write to scratchpad
       Case 0    ' Case SQLUDF_TF_FETCH:
         intRow = scratchPad(0)
         If intRow > staff.Length
            sqlState = "02000"  ' Return an error SQLSTATE
         Else
            ' Generate a row in the output table
            ' based on the staff array data.
            name = staff(intRow).GetName()
            position = staff(intRow).GetPosition()
            salary = (staff(intRow).GetSalary()) * factor
            nameNullInd = 0
            positionNullInd = 0
            salaryNullInd = 0
         End If
         intRow = intRow + 1
         scratchPad(0) = intRow  ' Write scratchpad

       Case 1    ' Case SQLUDF_TF_CLOSE:

       Case 2    ' Case SQLUDF_TF_FINAL:
    End Select

  End Sub

End Class
```

Example 2: Visual Basic parameter style SQL scalar function:
This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL scalar function
- Visual Basic code for a parameter style SQL scalar function

This scalar function returns a single count value for each input value that it operates on. For an input value in the nth position of the set of input values, the output scalar value is the value n. On each call of the scalar function, where one call is associated with each row or value in the input set of rows or values, the count is increased by one and the current value of the count is returned. The count is then saved in the scratchpad memory buffer to maintain the count value between each call of the scalar function.

This scalar function can be easily invoked if for example we have a table defined as follows:

```
CREATE TABLE T (i1 INTEGER);
INSERT INTO T VALUES 12, 45, 16, 99;
```

A simple query such as the following can be used to invoke the scalar function:

```
SELECT my_count(i1) as count, i1 FROM T;
```

The output of such a query would be:

```
COUNT          I1
-----------    ----------
1              12
2              45
3              16
4              99
```

This scalar UDF is quite simple. Instead of returning just the count of the rows, you could use a scalar function to format data in an existing column. For example you might append a string to each value in an address column or you might build up a complex string from a series of input strings or you might do a complex mathematical evaluation over a set of data where you must store an intermediate result.

Table 24. Code to create a Visual Basic parameter style SQL scalar function

```
CREATE FUNCTION mycount(INTEGER)
RETURNS INTEGER
LANGUAGE CLR
PARAMETER STYLE SQL
FENCED
SCRATCHPAD 10
FINAL CALL
VARIANT
NO SQL
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp';
```

Table 24. Code to create a Visual Basic parameter style SQL scalar function (continued)

```
Class empOps
  Public Shared Sub CountUp(byVal input As Int32, _
                            byRef outCounter As Int32, _
                            byVal nullIndInput As Int16, _
                            byRef nullIndOutCounter As Int16, _
                            byRef sqlState As String, _
                            byVal qualName As String, _
                            byVal specName As String, _
                            byRef sqlMessageText As String, _
                            byVal scratchPad As Byte(), _
                            byVal callType As Int32)

     Dim counter As Int32
     counter = 1

     Select callType
        case -1           ' case SQLUDF_TF_OPEN_CALL
           scratchPad(0) = counter
           outCounter = counter
           nullIndOutCounter = 0
        case 0              'case SQLUDF_TF_FETCH_CALL:
           counter = scratchPad(0)
           counter = counter + 1
           outCounter = counter
           nullIndOutCounter = 0
           scratchPad(0) = counter
        case 1             'case SQLUDF_CLOSE_CALL:
           counter = scratchPad(0)
           outCounter = counter
           nullIndOutCounter = 0
        case Else          ' Should never enter here
           ' These cases won't occur for the following reasons:
           ' Case -2  (SQLUDF_TF_FIRST)     ->No FINAL CALL in CREATE stmt
           ' Case 2   (SQLUDF_TF_FINAL)     ->No FINAL CALL in CREATE stmt
           ' Case 255 (SQLUDF_TF_FINAL_CRA) ->No SQL used in the function
           '
           ' * Note!*
           ' ---------
           ' The Else is required so that at compile time
           ' out parameter outCounter is always set *
           outCounter = 0
           nullIndOutCounter = -1
     End Select
  End Sub

End Class
```

Related concepts:

- "Common language runtime (CLR) routines" on page 106
- "User-defined scalar functions" on page 13
- "User-defined scalar functions" on page 15

Related tasks:

- "Examples of CLR procedures in Visual Basic" on page 130
- "Creating CLR routines" on page 107
- "Building Common Language Runtime (CLR) .NET routines" in the *Application Development Guide: Building and Running 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 identifying 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 154
- "SQL data type handling in C/C++ routines" on page 158

Related tasks:

- "Building UNIX C routines" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX C++ routines" in the *Application Development Guide: Building and Running Applications*
- "Building C/C++ routines on Windows" 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 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 89

Related samples:

- "spserver.c -- Definition of various types of stored procedures"
- "udfcli.c -- How to work with different types of user-defined functions (UDFs)"
- "spserver.sqC -- Definition of various 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 158
- "C/C++ routines" on page 151

Related reference:
- "Syntax for passing arguments to routines written in C/C++, OLE, or COBOL" on page 89
- "Supported SQL data types in C/C++" on page 155

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 25. 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

Table 25. SQL Data Types Mapped to C/C++ Declarations (continued)

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

DATE	char[11] SQLUDF_DATE	Null-terminated character string of the following format: `yyyy-mm-dd`

Table 25. SQL Data Types Mapped to C/C++ Declarations (continued)

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

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 154
- "SQL data type handling in C/C++ routines" on page 158
- "C/C++ routines" on page 151

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 might 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, DB2 moves *n* bytes of data to the buffer and sets the byte in the *n+1*> position to the null terminator (X'00'). For a RETURNS CHAR(n) value or an output parameter of a stored procedure that is not specified as FOR BIT DATA DB2 looks for a null terminator within the first *n* bytes of the CHAR value. If a null terminator is found, DB2 pads the remaining bytes, up to byte *n*, with ascii blanks. For a RETURNS CHAR(n) value or an output parameter of a stored procedure that is specified as FOR BIT DATA, DB2 copies over the first *n* bytes regardless of any occurrences of string null terminators within the *n* bytes. The string null terminators are treated as normal data.

  Exercise caution when using the normal C string handling functions in a routine that manipulates a FOR BIT DATA value, because many of these functions look for a null terminator to delimit a string argument and null terminators (X'00') can legitimately appear in the middle of a FOR BIT DATA value. Using the C functions on FOR BIT DATA values might cause the undesired truncation 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 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 operating systems 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 operating systems 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 operating systems 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 operating systems 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 can 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 can 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 284
- “Graphic host variables in C/C++ routines” on page 165
- “Include file for C/C++ routines (sqludf.h)” on page 154
- “C/C++ routines” on page 151

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 155

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 operating system, 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® operating systems, 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:

- "Parameter styles for external routines" on page 87
- "C/C++ routines" on page 151
- "Parameter handling in PROGRAM TYPE MAIN or PROGRAM TYPE SUB procedures" on page 51

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 administration notification 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 333
- "Table function execution model for Java" on page 59

Related tasks:

- "Debugging Java stored procedures" on page 175
- "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 170
- "JAR file administration on the database server" on page 173
- "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)"

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 26. SQL Data Types Mapped to Java Declarations

SQL Column Type	Java Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short, boolean	16-bit, signed integer
INTEGER (496 or 497)	int	32-bit, signed integer
BIGINT [1] (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
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	26-byte character string
GRAPHIC(*n*) (468 or 469)	java.lang.String	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	java.lang.String	Non-null-terminated varying double-byte character string with 2-byte string length indicator
LONGVARGRAPHIC (472 or 473)	java.lang.String	Non-null-terminated varying double-byte character string with 2-byte string length indicator

Note:

1. For Java applications connected from a DB2 UDB Version 8.1 client to a DB2 UDB Version 7.1 (or 7.2) server, note the following: when the getObject() method is used to retrieve a BIGINT value, a java.math.BigDecimal object is returned.

Where to put Java classes

You can use individual Java™ class files for your stored procedures and UDFs, or collect the class files into JAR files and install the JAR file in the database. If you decide to use JAR files, see the description of registering Java functions and stored procedures for more information.

Note: If you update or replace Java routine class files, you must issue a CALL SQLJ.REFRESH_CLASSES() statement to enable DB2® to load the updated classes. For more information on the CALL SQLJ.REFRESH_CLASSES() statement, see the description of how to update Java classes for routines.

To enable DB2 to find and use your Java language routines (stored procedures, UDFs or methods), you must store the corresponding class files as follows:

Unix and Windows® operating systems
In any directory path specified by your CLASSPATH variable. It is recommended that you store Java class files associated with DB2 routines in the *function directory*, `/u/`*`$DB2INSTANCE`*`/sqllib/function` where `/u/$DB2INSTANCE` is the directory associated with the currently active database manager.

The JVM that DB2 invokes uses the CLASSPATH environment variable to locate Java files. DB2 automatically adds the function directory and sqllib/java/db2java.zip to the front of your CLASSPATH setting so that you do not have to do this manually. It is recommended that you store your Java class files associated with DB2 routines in the function directory. Classes associated with unfenced routines should be stored in the `/u/`*`$DB2INSTANCE`*`/sqllib/function/unfenced` sub-directory.

To set your environment so that DB2 can find the JVM, you can set the *jdk_path* configuration parameter or use the default value. Also, you might need to set the *java_heap_sz* configuration parameter to increase the heap size for your application.

Note: If you declare a class to be part of a Java package, create subdirectories in the function directory that correspond to the fully qualified class names and place the related class files in the corresponding subdirectory. For example, if you create a class `ibm.tests.test1` for a Linux system, store the corresponding Java bytecode file (named test1.class) in sqllib/function/ibm/tests.

Related tasks:

- “Updating Java routines (stored procedures, UDFs, and methods) for runtime” on page 173

Related reference:

- “java_heap_sz - Maximum Java interpreter heap size configuration parameter” in the *Administration Guide: Performance*
- “jdk_path - Software Developer's Kit for Java installation path configuration parameter” in the *Administration Guide: Performance*

Updating Java routines (stored procedures, UDFs, and methods) for runtime

Procedure:

When you update Java routine classes, you must also issue a CALL SQLJ.REFRESH_CLASSES() statement to force DB2 to load the new classes. If you do not issue the CALL SQLJ.REFRESH_CLASSES() statement after you update Java routine classes, DB2 continues to use the previous versions of the classes. The CALL SQLJ.REFRESH_CLASSES() statement only applies to FENCED routines. DB2 refreshes the classes when a COMMIT or ROLLBACK occurs.

Note: You cannot update NOT FENCED routines without stopping and restarting the database manager.

Related concepts:
- "Java routines" on page 167

Related reference:
- "Java debug table DB2DBG.ROUTINE_DEBUG" on page 178
- "Java classes for DB2GENERAL routines" on page 337

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 )
```

Note: The privileges held by the authorization ID of the caller of sqlj.install_jar must include at least one of the following:
- CREATEIN privilege for the implicitly or explicitly specified schema
- SYSADM or DBADM authority

Replace

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

Remove

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

- *jar-url*: 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" on page 172
- "Java routines" on page 167
- "Library and class management considerations" on page 27

Connection contexts in SQLJ routines

With the introduction of multithreaded routines in DB2® Universal Database, Version 8, it is important that SQLJ routines avoid the use of the default connection context. That is, each SQL statement must explicitly indicate the ConnectionContext object, and that context must be explicitly instantiated in the Java™ method. For instance, in previous releases of DB2, a SQLJ routine could be written as follows:

```
class myClass
{
  public static void myRoutine( short myInput )
  {
    DefaultContext ctx = DefaultContext.getDefaultContext();
    #sql { some SQL statement };
  }
}
```

This use of the default context causes all threads in a multithreaded environment to use the same connection context, which, in turn, will result in unexpected failures.

The SQLJ routine above must be changed as follows:

```
#context MyContext;

class myClass
{
  public static void myRoutine( short myInput )
  {
    MyContext ctx = new MyContext( "jdbc:default:connection", false );
    #sql [ctx] { some SQL statement };
    ctx.close();
  }
}
```

This way, each invocation of the routine will create its own unique ConnectionContext (and underlying JDBC connection), which avoids unexpected interference by concurrent threads.

Related concepts:
- "Java routines" on page 167
- "Basic steps in writing an SQLJ application" in the *Application Development Guide: Programming Client Applications*
- "SQL statements in an SQLJ application" in the *Application Development Guide: Programming Client Applications*

Debugging stored procedures in Java

The following sections describe how to debug Java stored procedures.

Debugging Java stored procedures

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

The DB2 Distributed Debugger 9.2 must be properly configured to enable the debugging of Java Stored Procedures while working with DB2. The Distributed Debugger is included with all DB2 UDB Version 8 packaging options. The Distributed Debugger is configured to work with the standard SDK 1.3.1 level that is installed with DB2 UDB Version 8. If you are using a different SDK level you must update the DB2 database manager configuration with the following command from a DB2 command prompt:

```
db2 update dbm cfg using jdk_path <jdk131 path>
```

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 175
- "Populating the debug table" on page 177
- "Invoking the debug program" on page 177
- "Debugging routines" on page 38

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 SDK documentation.
2. Prepare the server.

   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.
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. Configure the Distributed Debugger:
   a. From a DOS command prompt, enter the following DB2SET command: `db2set DB2ROUTINE_DEBUG=on`.
   b. If operating on a UNIX operating system, complete the following steps (Windows users skip this step and continue with step c):
      1) `mkdir sqllib/function/src`
      2) `chmod 777 sqllib/function/src`
      3) `chmod 777 /home/youruserid/.DbgProf (some later Distributed Debugger versions such as 9.2.3)`
   c. Start DB2 (or restart DB2 if it is already started) by entering the following command from the command prompt: `db2start`
   d. Start the client daemon by entering the following command from the command prompt:

      ```
      idebug -qdaemon -quiport=portno
      ```

      where *quiport* is an unused TCP/IP port number. If you do not supply a value, the debug program uses 8000 as the default port number.

You are now ready to populate the debug table.

Related concepts:

- "Where to put Java classes" on page 172

Related tasks:

- "Populating the debug table" on page 177
- "Invoking the debug program" on page 177

- “Debugging routines” on page 38

Related reference:
- “Java debug table DB2DBG.ROUTINE_DEBUG” on page 178
- “JAR file administration on the database server” on page 173

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.

To start debugging Java Stored Procedures in the Development Center, use the Wizard to create a new Java Stored Procedure. From the options panel, select **enable debugging**.

To debug existing Java Stored Procedures that were previously not built with the **enable debugging** option:
1. From the Stored Procedures folder, right click and select **Build for Debug**.
2. Run the Stored Procedure in Debug mode by selecting the **Run/Debug** icon from the tool bar.

For more information on operating the Distributed Debugger, see the online help within the Distributed Debugger product.

Related tasks:
- “Preparing to debug Java stored procedures” on page 175
- “Populating the debug table” on page 177
- “Debugging routines” on page 38

Related reference:
- “Java debug table DB2DBG.ROUTINE_DEBUG” on page 178

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 175
- “Invoking the debug program” on page 177
- “Debugging routines” on page 38

Related reference:

- “Java debug table DB2DBG.ROUTINE_DEBUG” on page 178

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 27. 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)

Table 27. DB2DBG.ROUTINE_DEBUG Table Definition (continued)

Column Name	Data Type	Attributes	Description
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 Java stored procedures" on page 175
- "Preparing to debug Java stored procedures" on page 175
- "Populating the debug table" on page 177
- "Invoking the debug program" on page 177
- "Debugging routines" on page 38

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 181
- "OLE automation routines in BASIC and C++" on page 183

Related tasks:
- "Creating OLE automation routines" on page 180

Related reference:
- "Supported SQL data types in OLE automation" on page 182

Creating 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 create the methods of the object as routines using the CREATE statement. Creating 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 179
- "Object instance and scratchpad considerations and OLE routines" on page 181
- "OLE automation routines in BASIC and C++" on page 183

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 182

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 can 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 179
- "OLE automation routines in BASIC and C++" on page 183

Related tasks:
- "Creating OLE automation routines" on page 180

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 182

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 28. 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 28. 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 179
- "Object instance and scratchpad considerations and OLE routines" on page 181
- "OLE automation routines in BASIC and C++" on page 183

Related tasks:

- "Creating OLE automation routines" on page 180

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 29. 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

Table 29. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types (continued)

SQL Type	OLE Automation Type	BASIC Type	C++ Type
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 179
- "Object instance and scratchpad considerations and OLE routines" on page 181

Related tasks:
- "Creating OLE automation routines" on page 180

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 182

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.

Restrictions on using OLE DB table functions: OLE DB table functions cannot connect to a DB2 database.

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 189

Related tasks:
- "Creating an OLE DB table UDF" on page 187

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 190

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 189
- "OLE DB user-defined table functions" on page 186

Related tasks:
- "Installing DB2 Information Integrator and setting up the server to access OLE DB data sources" in the *IBM DB2 Information Integrator Installation Guide*
- "Adding OLE DB data sources to a federated server" in the *IBM DB2 Information Integrator Data Source Configuration 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 190

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 186

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 30. 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

Table 30. Mapping DB2 Data Types to OLE DB (continued)

DB2 Data Type	OLE DB Data Type
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

Note: OLE DB data type conversion rules are defined in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. For example:

- To retrieve the OLE DB data type DBTYPE_CY, the data 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 186

Related tasks:

- "Creating an OLE DB table UDF" on page 187

Chapter 5. Invoking routines

Routine invocation

Once a routine has been developed and created in the database by issuing the CREATE statement, if the appropriate routine priviliges have been granted to the routine definer and routine invoker, the routine can be invoked.

Each routine type serves a different purpose and is used in a different way. The prerequisites for invoking routines is common, but the implementation of the invocation differs for each.

Prerequisites for routine invocation:

- The routine must have been created in the database using the CREATE statement.
- For an external routine, the library or class file must be installed in location specified by the EXTERNAL clause of the CREATE statement, or an error (SQLCODE SQL0444, SQLSTATE 42724) will occur.
- The routine invoker must have the EXECUTE privilege on the routine. If the invoker is not authorized to execute the routine, an error (SQLSTATE 42501) will occur.

Procedure invocation:

Procedures are invoked by executing the CALL statement with a reference to a procedure.

The CALL statement enables the procedure invocation, the passing of parameters to the procedure, and the receiving of parameters returned from the procedure. Any accessible result sets returned from a procedure can be processed once the procedure has successfully returned.

Procedures can be invoked from anywhere that the CALL statement is supported including:

- client applications
- External routines (procedure, UDF, or method)
- SQL routines (procedure, UDF, or method)
- Triggers (before triggers, after triggers, or instead of triggers)
- Dynamic compound statements

- Command line processor (CLP)

If you choose to invoke a procedure from a client application or from an external routine, the client application or external routine can be written in a language other than that of the procedure. For example, a client application written in C++ can use the CALL statement to invoke a procedure written in Java™. This provides programmers with great flexibility to program in their language of choice and to integrate code pieces written in different languages.

In addition, the client application that invokes the procedure can be executed on a different operating system than the one where the procedure resides. For example a client application running on a Windows® operating system can use the CALL statement to invoke a procedure residing on a Linux database server.

Depending on where a procedure is invoked from there may be some additional considerations.

Function invocation:

Functions are intended to be referenced within SQL statements.

Built-in functions, sourced aggregate functions, and scalar user-defined can be referenced wherever an expression is allowed within an SQL statement. For example within the select-list of a query or within the VALUES clause of an INSERT statement. Table functions can only be referenced in the FROM clause. For example in the FROM clause of a query or a data change statement.

Method invocation:

Methods are similar to scalar functions except that they are used to give behavior to structured types. Method invocation is the same as scalar user-defined function invocation, except that one of the parameters to the method must be the structured type that the method operates on.

Routine invocation related-tasks:

To invoke a particular type of routine:
- "Calling procedures from applications or external routines" on page 200
- "Calling procedures from triggers or SQL routines" on page 202
- Call a procedure from a CLI application
- Call a procedure from the Command line processor (CLP)
- "Invoking scalar functions or methods" on page 211
- "Invoking user-defined table functions" on page 212

Related concepts:
- "Routines in application development" on page 3
- "Routine code page considerations" on page 197
- "Routine names and paths" on page 195
- "Nested routine invocations" on page 196

Related tasks:
- "Writing routines" on page 33
- "Creating routines in the database" on page 31

| • "Invoking 32-bit routines on a 64-bit database server" on page 197

| **Related reference:**

| • "CALL statement" in the *SQL Reference, Volume 2*

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 can 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:
- "Routines in application development" on page 3
- "User-defined structured types" on page 245
- "Dynamic dispatch of methods" on page 251
- "Function selection" on page 208
- "Types of routines (procedures, functions, methods)" on page 5
- "Procedure selection" on page 200

Related tasks:
- "Defining behavior for structured types" on page 251

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*

Nested routine invocations

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 the series of routines that is referenced contains a routine that was previously referenced this is said to be 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.

Other 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:

- “Data conflicts when procedures read from or write to tables” on page 40
- “Security considerations for routines” on page 24

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 procedures, a 32-bit Java Virtual Machine (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 because the routine needs to run in FENCED NOT THREADSAFE mode. So, 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 invoke 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:

- “Routine invocation” on page 193
- “Java routines” on page 167

Routine code page considerations

Character data is passed to external routines in the code page implied by the PARAMETER CCSID option used when the routine was created. Similarly, a character string that is output from the routine is assumed by the database to use the code page implied by the PARAMETER CCSID option.

When a client program (using, for example, code page C) accesses a section with a different code page (for example, code page S) that invokes a routine using a different code page (for example, code page R), the following events occur:

1. When an SQL statement is invoked, input character data is converted from the code page of the client application (C) to the one associated with the section (S). Conversion does not occur for BLOBs or data that will be used as FOR BIT DATA.
2. If the code page of the routine is not the same as the code page of the section, then before the routine is invoked, input character data (except for BLOB and FOR BIT DATA) is converted to the code page of the routine (R).

   It is strongly recommended that you precompile, compile, and bind the server routine using the code page that the routine will be invoked under (R). 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 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 routine finishes, the database manager converts all output character data from the routine code page (R) to the section code page (S) if necessary. If the routine raised an error during its execution, the SQLSTATE and diagnostic message from the routine will also be converted from the routine code page to the section code page. Conversion does not happen for BLOB or FOR BIT DATA character strings.
4. When the statement finishes, output character data is converted from the section code page (S) back to code page of the client application (C). 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 routine 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*

Procedure invocation

References to 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 87

Related tasks:

- “Calling procedures from the Command Line Processor (CLP)” on page 204
- “Calling stored procedures from CLI applications” in the *CLI Guide and Reference, Volume 1*
- “Calling procedures from triggers or SQL routines” on page 202
- “Calling procedures from applications or external routines” on page 200

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 89

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:

- "Routine invocation" on page 193
- "References to procedures" on page 199

Related tasks:

- "Calling procedures from triggers or SQL routines" on page 202
- "Calling procedures from applications or external routines" on page 200

Calling procedures from applications or external routines

Invoking a procedure (also called a stored procedure) that encapsulates logic from a client application or from an application associated with an external routine is easily done with some simple setup work in the application and by using the CALL statement.

Prerequisites:

The procedure must have been created in the database by executing the CREATE PROCEDURE statement.

For external procedures, the library or class file must exist in the location specified by the EXTERNAL clause in the CREATE PROCEDURE statement.

The procedure invoker must have the privileges required to execute the CALL statement. The procedure invoker in this case is the user ID executing the application, however special rules apply if the DYNAMICRULES bind option is used for the application.

Procedure:

Certain elements must be included in your application if you want that application to invoke a procedure. In writing your application you must do the following:

1. Declare, allocate, and initialize storage for the optional data structures and host variables or parameter markers required for the CALL statement.

   To do this:

   - Assign a host variable or parameter marker to be used for each parameter of the procedure.
   - Initialize the host variables or parameter markers that correspond to IN or INOUT parameters.
2. Establish a database connection. Do this by executing an embedded SQL language CONNECT TO statement, or by coding an implicit database connection.
3. Code the procedure invocation. After the database connection code, you can code the procedure invocation. Do this by executing the SQL language CALL statement. Be sure to specify a host variable, constant, or parameter marker for each IN, INOUT, OUT parameter that the procedure expects.
4. Add code to process the OUT and INOUT parameters, and result sets. This code must come after the CALL statement execution.
5. Code a database COMMIT or ROLLBACK. Subsequent to the CALL statement and evaluation of output parameter values or data returned by the procedure, you might want your application to commit or roll back the transaction. This can be done by including a COMMIT or ROLLBACK statement. A procedure can include a COMMIT or ROLLBACK statement, however it is recommended practice that transaction management be done within the client application.

   Note: Procedures invoked from an application that established a type 2 connection to the database, cannot issue COMMIT or ROLLBACK statements.
6. Disconnect from the database.
7. Prepare, compile, link, and bind your application. If the application is for an external routine, issue the CREATE statement to create the routine and locate your external code library in the appropriate function path for your operating system so that the database manager can find it.
8. Run your application or invoke your external routine. The CALL statement that you embedded in your application will be invoked.

Note: You can code SQL statements and routine logic at any point between steps 2 and 5.

Related concepts:
- "Routine invocation" on page 193
- "Procedure selection" on page 200
- "References to procedures" on page 199

Related tasks:
- "Calling procedures from triggers or SQL routines" on page 202

Related reference:
- "COMMIT statement" in the *SQL Reference, Volume 2*
- "ROLLBACK statement" in the *SQL Reference, Volume 2*

Related samples:
- "spcall.c -- Call individual stored procedures"

- "spclient.c -- Call various stored procedures"
- "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)"

Calling procedures from triggers or SQL routines

Calling a procedure from an SQL routine, a trigger, or dynamic compound statement is essentially the same. The same steps are used to implement this call. This topic explains the steps using a trigger scenario. Any prerequisites or steps that differ when calling a procedure from a routine or dynamic compound statement are stated.

Prerequisites:

- The procedure must have been created in the database by executing the CREATE PROCEDURE statement.
- For external procedures, the library or class files must be in the location specified by the EXTERNAL clause of the CREATE PROCEDURE statement.
- The creator of a trigger that contains a CALL statement must have the privilege to execute the CALL statement. At runtime when a trigger is activated it is the authorization of the creator of the trigger that is checked for the privilege to execute the CALL statement. A user that executes a dynamic compound statement that contains a CALL statement, must have the privilege to execute the CALL statement for that procedure.
- To invoke a trigger, a user must have the privilege to execute the data change statement associated with the trigger event. Similarly, to successfully invoke an SQL routine or dynamic compound statement a user must have the EXECUTE privilege on the routine.

Restrictions:

When invoking a procedure from within an SQL trigger, an SQL routine, or a dynamic compound statement the following restrictions apply:

- In partitioned database environments procedures cannot be invoked from triggers or SQL UDFs.
- On symmetric multi-processor (SMP) machines, procedure calls from triggers are executed on a single processor.
- A procedure that is to be called from a trigger must not contain a COMMIT statement or a ROLLBACK statement that attempts to rollback the unit of work. The ROLLBACK TO SAVEPOINT statement is supported within the procedure however the specified savepoint must be in the procedure.
- A rollback of a CALL statement from within a trigger will not rollback the statements executed within the procedure.
- The procedure must not modify any federated table. This means that the procedure must not contain a searched UPDATE of a nickname, a searched DELETE from a nickname or an INSERT to a nickname.
- Result sets specified for the procedure will not be accessible.

BEFORE triggers can not be created if they contain a CALL statement that references a procedure created with an access level of MODIFIES SQL DATA. The

execution of a CREATE TRIGGER statement for such a trigger will fail with error (SQLSTATE 42987). For more about SQL access levels in routines see:
- “SQL access levels in SQL routines” on page 63
- “SQL in external routines” on page 101

Procedure:

This procedure section explains how to create and invoke a trigger that contains a CALL statement. The SQL required to call a procedure from a trigger is the same SQL required to call a procedure from an SQL routine or dynamic compound statement.

1. Write a basic CREATE TRIGGER statement specifying the desired trigger attributes. See the CREATE TRIGGER statement.
2. In the trigger action portion of the trigger you can declare SQL variables for any IN, INOUT, OUT parameters that the procedure specifies. See the DECLARE statement. To see how to initialize or set these variables see the assignment statement. Trigger transition variables can also be used as parameters to a procedure.
3. In the trigger action portion of the trigger add a CALL statement for the procedure. Specify a value or expression for each of the procedure’s IN, INOUT, and OUT parameters
4. For SQL procedures you can optionally capture the return status of the procedure by using the GET DIAGNOSTICS statement. To do this you will need to use an integer type variable to hold the return status. Immediately after the CALL statement, simply add a GET DIAGNOSTICS statement that assigns RETURN_STATUS to your local trigger return status variable.
5. Having completed writing your CREATE TRIGGER statement you can now execute it statically (from within an application) or dynamically (from the CLP, or from the Control Center) to formally create the trigger in the database.
6. Invoke your trigger. Do this by executing against the appropriate data change statement that corresponds to your trigger event.
7. When the data change statement is executed against the table, the appropriate triggers defined for that table are fired. When the trigger action is executed, the SQL statements contained within it, including the CALL statement, are executed.

Run-time errors:

If the procedure attempts to read or write to a table that the trigger also reads or writes to, an error might be raised if a read or write conflict is detected. The set of tables that the trigger modifies, including the table for which the trigger was defined must be exclusive from the tables modified by the procedure.

Example: Calling an SQL procedure from a trigger:

This example illustrates how you can embed a CALL statement to invoke a procedure within a trigger and how to capture the return status of the procedure call using the GET DIAGNOSTICS statement. The SQL below creates the necesary tables, an SQL PL language procedure, and an after trigger.

```
CREATE TABLE T1 (c1 INT, c2 CHAR(2))@
CREATE TABLE T2 (c1 INT, c2 CHAR(2))@

CREATE PROCEDURE proc(IN val INT, IN name CHAR(2))
LANGUAGE SQL
```

```
DYNAMIC RESULTSETS 0
MODIFIES SQL DATA
BEGIN
  DECLARE rc INT DEFAULT 0;
  INSERT INTO TABLE T2 VALUES (val, name);
  GET DIANOSTICS rc = ROW_COUNT;
  IF ( rc > 0 ) THEN
      RETURN 0;
  ELSE
      RETURN -200;
  END IF;
END@

CREATE TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (n.c1 > 100);
BEGIN ATOMIC
   DECLARE rs INTEGER DEFAULT 0;
   CALL proc(n.c1, n.c2);
   GET DIANOSTICS rs = RETURN_STATUS;
   VALUES(CASE WHEN rc < 0 THEN RAISE_ERROR('70001', 'PROC CALL failed'));
END@
```

Issuing the following SQL statement will cause the trigger to fire and the procedure will be invoked.

```
UPDATE T1 SET c1 = c1+1 WHERE c2 = 'CA'@
```

Related concepts:
- "SQL access levels in SQL routines" on page 63
- "SQL in external routines" on page 101
- "Routine invocation" on page 193
- "Procedure selection" on page 200
- "References to procedures" on page 199

Related tasks:
- "Calling stored procedures from CLI applications" in the *CLI Guide and Reference, Volume 1*
- "Calling procedures from applications or external routines" on page 200

Related reference:
- "CALL statement" in the *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in the *SQL Reference, Volume 2*
- "CREATE TRIGGER statement" in the *SQL Reference, Volume 2*
- "GET DIAGNOSTICS statement" in the *SQL Reference, Volume 2*

Calling procedures from the Command Line Processor (CLP)

You can call stored procedures by using the `CALL` statement from the DB2 command line processor interface. The stored procedure being called must be defined in the DB2 system catalog tables.

Procedure:

To call the stored procedure, first connect to the database:

```
   db2 connect to sample user userid using password
```

where *userid* and *password* are the user ID and password of the instance where the `sample` database is located.

To use the `CALL` statement, enter the stored procedure name plus any IN or INOUT parameter values, as well as '?' as a place-holder for each OUT parameter value.

The parameters for a stored procedure are given in the `CREATE PROCEDURE` statement for the stored procedure in the program source file.

SQL procedure example

For information on creating an SQL procedure, see 'Creating SQL procedures' in Chapter 6. SQL procedures.

In the `whiles.db2` file, the `CREATE PROCEDURE` statement for the `DEPT_MEDIAN` procedure signature is as follows:

```
CREATE PROCEDURE DEPT_MEDIAN
(IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
```

To invoke this procedure, use the CALL statement in which you must specify the procedure name and appropriate parameter arguments, which which in this are the value for the IN parameter, and a question mark, '?', for the value of the OUT parameter. The procedure's SELECT statement uses the `deptNumber` value on the DEPT column of the STAFF table, so to get meaningful output the IN parameter needs to be a valid value from the DEPT column; for example, the value "51":

```
db2 call dept_median (51, ?)
```

Note: On UNIX platforms the parentheses have special meaning to the command shell, so they must be preceded with a "\" character or surrounded with quotes, as follows:

```
db2 "call dept_median (51, ?)"
```

You do not use quotes if you are using the interactive mode of the command line processor.

After running the above command, you should receive this result:

```
Value of output parameters
--------------------------
Parameter Name  : MEDIANSALARY
Parameter Value : +1.76545000000000E+004

Return Status = 0
```

C stored procedure example

You can also call stored procedures created from supported host languages with the Command Line Processor. In the `samples/c` directory on UNIX, and the `samples\c` directory on Windows, DB2 provides files for creating stored procedures. The `spserver` shared library contains a number of stored procedures that can be created from the source file, `spserver.sqc`. The `spcreate.db2` file catalogs the stored procedures.

In the `spcreate.db2` file, the `CREATE PROCEDURE` statement for the `MAIN_EXAMPLE` procedure begins:

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
                               OUT salary DOUBLE,
                               OUT errorcode INTEGER)
```

To call this stored procedure, you need to put in a CHAR value for the IN parameter, `job`, and a question mark, '?', for each of the OUT parameters. The procedure's SELECT statement uses the `job` value on the JOB column of the EMPLOYEE table, so to get meaningful output the IN parameter needs to be a valid value from the JOB column. The C sample program, `spclient`, that calls the stored procedure, uses `'DESIGNER'` for the `JOB` value. We can do the same, as follows:

```
db2 "call MAIN_EXAMPLE ('DESIGNER', ?, ?)"
```

After running the above command, you should receive this result:

```
Value of output parameters
--------------------------
Parameter Name  : SALARY
Parameter Value : +2.37312500000000E+004

Parameter Name  : ERRORCODE
Parameter Value : 0

Return Status = 0
```

An `ERRORCODE` of zero indicates a successful result.

Comparing with the `spclient` program, we see that `spclient` has formatted the result in decimal for easier viewing:

```
CALL stored procedure named MAIN_EXAMPLE
Stored procedure returned successfully
Average salary for job DESIGNER =  23731.25
```

Related tasks:

- "Creating SQL procedures" in the *Application Development Guide: Building and Running Applications*
- "Calling SQL procedures with client applications" 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*
- "Calling procedures from triggers or SQL routines" on page 202
- "Calling procedures from applications or external routines" on page 200

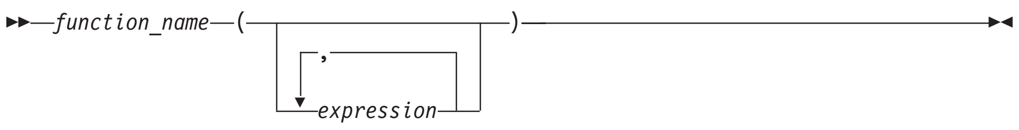
Related samples:

- "spclient.sqc -- Call various stored procedures (C)"
- "spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc (C)"
- "spserver.sqc -- Definition of various types of stored procedures (C)"
- "whiles.db2 -- To create the DEPT_MEDIAN SQL procedure "
- "whiles.sqc -- To call the DEPT_MEDIAN SQL procedure"

Function and method invocation

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 might 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 scalar functions or methods" on page 211
- "Invoking user-defined table functions" on page 212

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 206

Related tasks:
- "Invoking scalar functions or methods" on page 211
- "Invoking user-defined table functions" on page 212

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 UDF or method 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 can 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 can 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 208
- "Procedure selection" on page 200

Related tasks:
- "Passing structured type parameters to external routines" on page 292
- "LOB values as UDF parameters" on page 210

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 208
- "Procedure selection" on page 200

Related tasks:

- "Retrieving a LOB value with a LOB locator" on page 220
- "Distinct types as UDF or method parameters" on page 209

Related reference:

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

Invoking scalar functions or methods

The invocation of built-in scalar functions, user-defined scalar-functions and methods is very similar. Scalar functions and methods can only be invoked where expressions are supported within an SQL statement.

Prerequisites:

- For built-in functions, SYSIBM must be in the CURRENT PATH special register. SYSIBM is in CURRENT PATH by default.
- For user-defined scalar functions, the function must have been created in the database using either the CREATE FUNCTION or CREATE METHOD statement.
- For external user-defined scalar functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION or CREATE METHOD statement.
- To invoke a user-defined function or method, a user must have EXECUTE privilege on the function or method. If the function or method is to be used by all users, the EXECUTE privilege on the function or method can be granted to PUBLIC. For more privilege related information see the specific CREATE statement reference.

Procedure:

To invoke a scalar UDF or method:

- Include a reference to it within an expression contained in an SQL statement where it is to process one or more input values. Functions and methods can be invoked anywhere that an expression is valid. Examples of where a scalar UDF or method can be referenced include the select-list of a query or in a VALUES clause

For example, suppose that you have created a user-defined scalar function 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 206
- "Routine invocation" on page 193
- "Routine names and paths" on page 195
- "User-defined scalar functions" on page 13
- "Methods" on page 16

Related tasks:

- "Invoking user-defined table functions" on page 212

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.

Prerequisites:

- The table function must have been created in the database by executing the CREATE FUNCTION.
- For external user-defined table functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION.
- To invoke a user-defined table function a user must have EXECUTE privilege on the function. For more privilege related information see the CREATE FUNCTION reference.

Restrictions:

For restrictions on invoking user-defined table functions, see the CREATE FUNCTION topics in the related links.

Procedure:

To invoke a user-defined table function, reference the function in the FROM clause of an SQL statement where it is to process a set of input values. The reference to the table function must be preceeded by the TABLE clause and be contained in brackets.

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 206
- "Routine names and paths" on page 195
- "User-defined scalar functions" on page 15

Related tasks:
- "LOB values as UDF parameters" on page 210
- "Invoking scalar functions or methods" on page 211

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)”

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 could be sufficient for small to medium size text data, applications often need to store large text documents. They might also need to store a wide variety of additional data types, such as audio, video, drawings, mixed text and graphics, and images. DB2® provides three data types to store these data objects as strings of up to 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 can have a large amount of associated LOB data. Although any single LOB value cannot exceed 2 gigabytes, a single row can contain as much as 24 gigabytes of LOB data, and a table can 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. Storing large object data outside their records 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 for CREATE TABLE gives you the choice to log (or not) the changes made to the LOB columns. 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 for CREATE TABLE allows you to identify a LARGE
| 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 1 GB 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 the following options:

- 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 might not have the capacity to hold larger LOB values.
- Select only 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.
- 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 218
- "Large object file reference variables" on page 223

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 is 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 217

Related tasks:

- "Retrieving a LOB value with a LOB locator" on page 220
- "Deferring the evaluation of LOB expressions" on page 221

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. Connect to 'sample' database using JDBC type 2 driver (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 217
- "Large object locators" on page 218

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 221

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. Connect to 'sample' database using JDBC type 2 driver (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 217
- "Large object locators" on page 218

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 220

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. Connect to 'sample' database using JDBC type 2 driver (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 can 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.
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 217

Related tasks:
- "Writing data from a CLOB column to a text file" on page 225
- "Inserting data from a text file into a CLOB column" on page 226

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. Connect to 'sample' database using JDBC type 2 driver (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 218
- "Large object file reference variables" on page 223

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 226

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. Connect to 'sample' database using JDBC type 2 driver (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 218
- "Large object file reference variables" on page 223

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 225

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. Connect to 'sample' database using JDBC type 2 driver (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 229
- "User-defined structured types" on page 245

Related tasks:
- "Creating distinct types" on page 231

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 components such as built-in functions, comparison operators, and indexes 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 231
- "User-defined types" on page 229

Related tasks:

- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Manipulating distinct types" on page 237

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 USER DEFINED DISTINCT TYPES (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233

Creating 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 231
- "User-defined types" on page 229
- "User-defined distinct types" on page 229

Related tasks:
- "Creating currency-based distinct types" on page 235
- "Creating a distinct type for completed job application forms" on page 235
- "Manipulating distinct types" on page 237

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 USER DEFINED DISTINCT TYPES (JDBC)"

- "DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (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 231
- "User-defined types" on page 229
- "User-defined distinct types" on page 229

Related tasks:

- "Creating tables to track international sales" on page 236
- "Creating a table to store completed job application forms" on page 237
- "Creating distinct types" on page 231
- "Manipulating distinct types" on page 237
- "Creating currency-based distinct types" on page 235
- "Creating a distinct type for completed job application forms" on page 235

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 229
- “User-defined distinct types” on page 229
- “User-defined structured types” on page 245
- “Transform functions and transform groups” on page 284

Related tasks:

- “Creating distinct types” on page 231
- “Creating structured types” on page 246

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 USER DEFINED DISTINCT TYPES (JDBC)”
- “DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (SQLJ)”
- “DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)”

Creating 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 229

Related tasks:

- "Creating tables with columns based on distinct types" on page 233
- "Creating a distinct type for completed job application forms" on page 235
- "Creating tables to track international sales" on page 236

Related reference:

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

Creating 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 229

Related tasks:
- "Creating tables with columns based on distinct types" on page 233
- "Creating currency-based distinct types" on page 235
- "Creating a table to store completed job application forms" on page 237

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating currency-based distinct types" on page 235
- "Creating a table to store completed job application forms" on page 237

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 229

Related tasks:

- "Creating distinct types" on page 231
- "Creating a distinct type for completed job application forms" on page 235
- "Creating tables to track international sales" on page 236

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 229

Related tasks:
- "Casting between distinct types" on page 238
- "Performing comparisons involving distinct types" on page 239
- "Performing comparisons between distinct types and constants" on page 240
- "Defining sourced UDFs for distinct types" on page 243
- "Performing assignments involving distinct types in embedded SQL" on page 240
- "Performing assignments involving distinct types in dynamic SQL" on page 241
- "Performing assignments involving different distinct types" on page 241
- "Performing UNION operations on distinctly typed columns" on page 242
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233

Related samples:
- "dtudt.c -- How to create, use, and drop user-defined distinct types."
- "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 USER DEFINED DISTINCT TYPES (JDBC)"
- "DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (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 229

Related tasks:

- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Defining sourced UDFs for distinct types" on page 243

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Casting between distinct types" on page 238

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Casting between distinct types" on page 238

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Defining sourced UDFs for distinct types" on page 243

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233

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 229

Related tasks:

- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Casting between distinct types" on page 238

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 229

Related tasks:
- "Creating distinct types" on page 231
- "Creating tables with columns based on distinct types" on page 233
- "Performing assignments involving distinct types in embedded SQL" on page 240

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 can 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 229
- “Typed tables” on page 255
- “Typed views” on page 269

Related tasks:
- “Creating structured types” on page 246
- “Storing instances of structured types” on page 247
- “Defining behavior for structured types” on page 251
- “Dropping user-defined types” on page 234

Related samples:
- “dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)”
- “dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)”

Creating 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 229
- "User-defined structured types" on page 245
- "Structured type hierarchies" on page 248

Related tasks:
- "Storing instances of structured types" on page 247
- "Creating a structured type hierarchy" on page 249
- "Dropping user-defined types" on page 234

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 245

- “Typed tables” on page 255

Related tasks:

- “Storing objects in typed table rows” on page 260
- “Storing structured type objects in table columns” on page 276

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. Conversely, noninstantiable types 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 245

Related tasks:

- “Creating structured types” on page 246

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`.

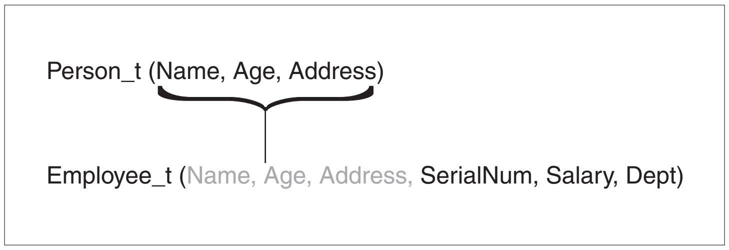

Person_t (Name, Age, Address)
Employee_t (Name, Age, Address, SerialNum, Salary, Dept)

Figure 3. 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:

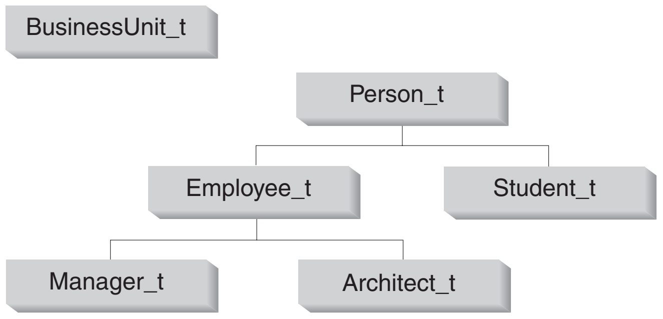


Figure 4. Type hierarchies (BusinessUnit_t and Person_t)

In Figure 4, 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 245

Related tasks:

- "Creating structured types" on page 246
- "Creating a structured type hierarchy" on page 249

Creating a structured type hierarchy

The following figure presents an illustration of a structured type hierarchy:

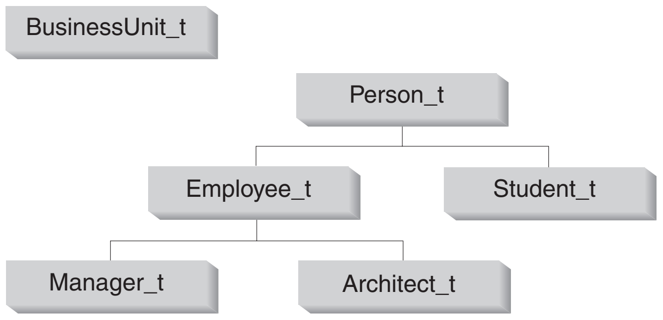


Figure 5. Type hierarchies (BusinessUnit_t and Person_t)

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 245
- "Structured type hierarchies" on page 248

Related tasks:

- "Creating structured types" on page 246
- "Creating typed tables" on page 255

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 245
- "Dynamic dispatch of methods" on page 251

Related tasks:
- "Creating structured types" on page 246

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 245
- "Structured type hierarchies" on page 248

Related tasks:
- "Creating structured types" on page 246
- "Defining behavior for structured types" on page 251

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 245
- "Reference types" on page 264

Related tasks:
- "Creating structured types" on page 246

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 245

Related tasks:

- "Creating structured types" on page 246

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 245

Related tasks:

- "Creating structured types" on page 246

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 245

Related tasks:
- "Creating structured types" on page 246

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 245
- "Reference types" on page 264
- "Substitutability in typed tables" on page 259
- "Typed views" on page 269

Related tasks:
- "Storing objects in typed table rows" on page 260
- "Dropping typed tables" on page 258
- "Defining system-generated object identifiers" on page 261
- "Defining constraints on object identifier columns" on page 263
- "Creating typed tables" on page 255

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.

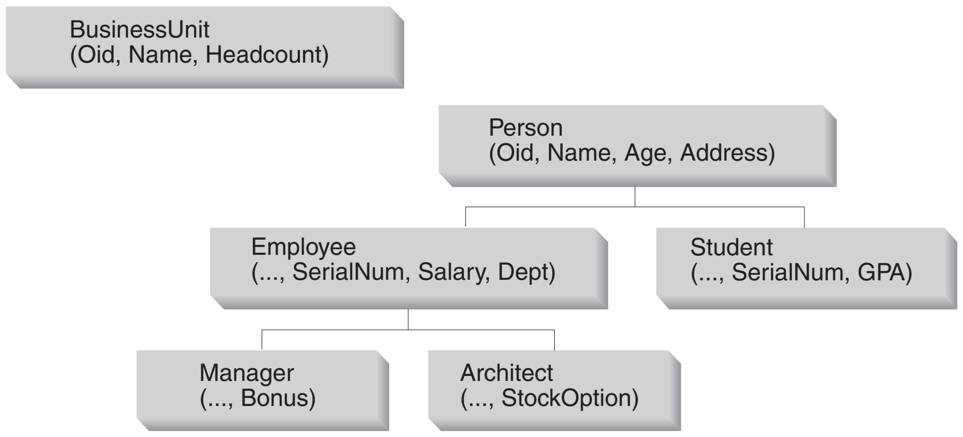


Figure 6. Typed table hierarchy

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 can 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 245
- "Structured type hierarchies" on page 248
- "Typed tables" on page 255

Related tasks:
- "Creating structured types" on page 246
- "Storing objects in typed table rows" on page 260
- "Dropping typed tables" on page 258
- "Defining system-generated object identifiers" on page 261
- "Defining constraints on object identifier columns" on page 263

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 can 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 contsraints.

Related concepts:

- "Structured type hierarchies" on page 248
- "Typed tables" on page 255

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 248
- "Typed tables" on page 255

Related tasks:
- "Creating a structured type hierarchy" on page 249
- "Issuing queries to dereference references" on page 272

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 31. 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 32. 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 33. 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 34. 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 265
- “Substitutability in typed tables” on page 259
- “Typed tables” on page 255

Related tasks:

- “Storing instances of structured types” on page 247
- “Creating typed tables” on page 255

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 264
- "Relationships between objects in typed tables" on page 265

Related tasks:

- "Creating a structured type hierarchy" on page 249
- "Issuing queries to dereference references" on page 272
- "Creating typed tables" on page 255

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 7 on page 264.

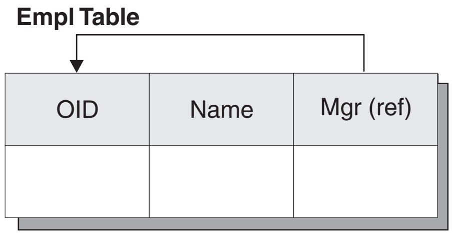


Figure 7. Self-referencing type example

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 264

Related tasks:

- "Creating structured types" on page 246
- "Storing objects in typed table rows" on page 260
- "Defining system-generated object identifiers" on page 261

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 268
- "Relationships between objects in typed tables" on page 265
- "Typed tables" on page 255

Related tasks:

- "Storing objects in typed table rows" on page 260
- "Issuing queries to dereference references" on page 272
- "Defining system-generated object identifiers" on page 261
- "Defining constraints on object identifier columns" on page 263
- "Creating typed tables" on page 255

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 8.

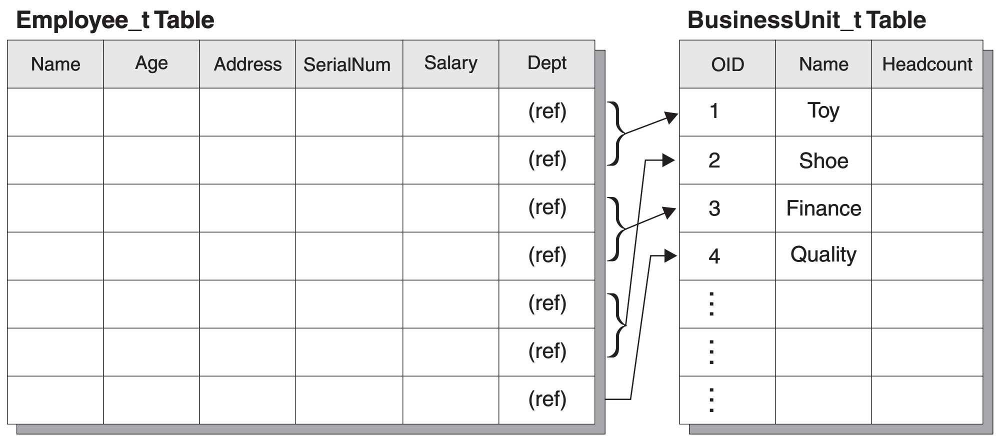


Figure 8. Structured type references from Employee_t to BusinessUnit_t

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 264
- "Referential integrity versus scoped references" on page 268
- "Typed tables" on page 255

Related tasks:

- "Restricting returned types using a TYPE predicate" on page 275
- "Defining system-generated object identifiers" on page 261

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.

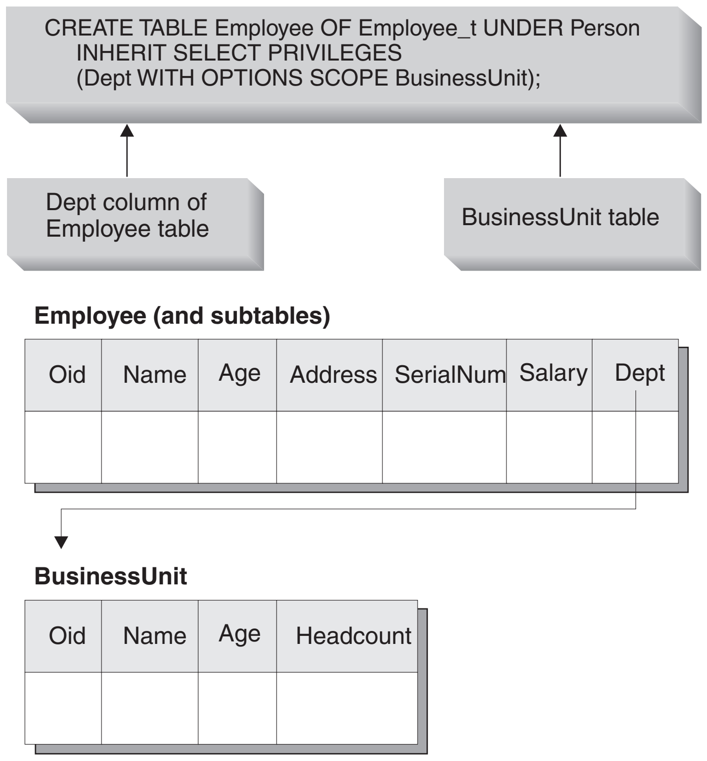

CREATE TABLE Employee OF Employee_t UNDER Person
INHERIT SELECT PRIVILEGES
(Dept WITH OPTIONS SCOPE BusinessUnit);
Dept column of Employee table
BusinessUnit table
Employee (and subtables)
Oid
Name
Age
Address
SerialNum
Salary
Dept
BusinessUnit
Oid
Name
Age
Headcount

Figure 9. 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)
```

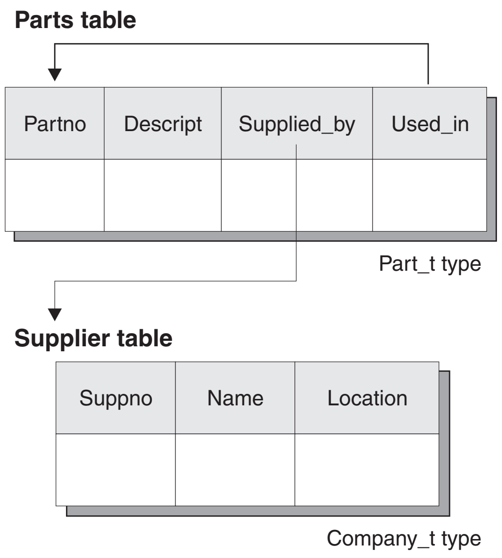


Figure 10. Example of a self-referencing scope

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 264
- “Referential integrity versus scoped references” on page 268
- “Relationships between objects in typed tables” on page 265
- “Typed tables” on page 255

Related tasks:
- “Defining system-generated object identifiers” on page 261

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 264
- “Typed tables” on page 255

Related tasks:
- "Defining semantic relationships with references" on page 266

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 245
- "Typed tables" on page 255

Related tasks:
- "Creating typed views" on page 269
- "Altering typed views" on page 271
- "Dropping typed views" on page 272

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 245
- "Typed tables" on page 255
- "Typed views" on page 269

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 245
- "Typed tables" on page 255
- "Typed views" on page 269

Related tasks:
- "Creating typed views" on page 269

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 245
- “Typed tables” on page 255
- “Typed views” on page 269

Related tasks:

- “Creating typed views” on page 269

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 245
- "Reference types" on page 264
- "Relationships between objects in typed tables" on page 265
- "Typed tables" on page 255
- "Typed views" on page 269

Related tasks:
- "Storing objects in typed table rows" on page 260
- "Returning objects of a particular type using ONLY" on page 274
- "Restricting returned types using a TYPE predicate" on page 275
- "Returning all possible types using OUTER" on page 275
- "Defining system-generated object identifiers" on page 261

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 229
- "Typed tables" on page 255

Related tasks:
- "Storing objects in typed table rows" on page 260
- "Issuing queries to dereference references" on page 272

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 245
- "Reference types" on page 264
- "Typed tables" on page 255
- "Typed views" on page 269

Related tasks:

- "Storing objects in typed table rows" on page 260
- "Issuing queries to dereference references" on page 272

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 245
- "Typed tables" on page 255
- "Typed views" on page 269

Related tasks:

- "Storing structured type objects in table columns" on page 276
- "Issuing queries to dereference references" on page 272

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 can 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 11.

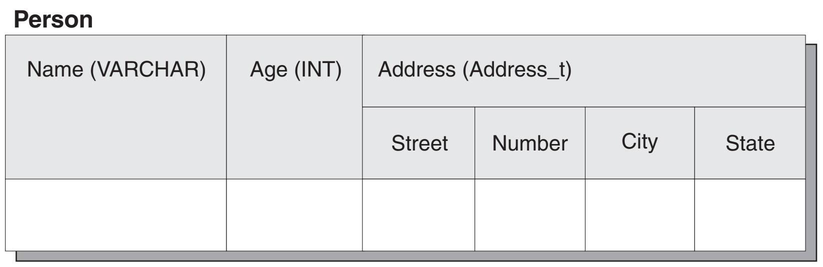


Figure 11. Address attribute as a structured type

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 12 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.

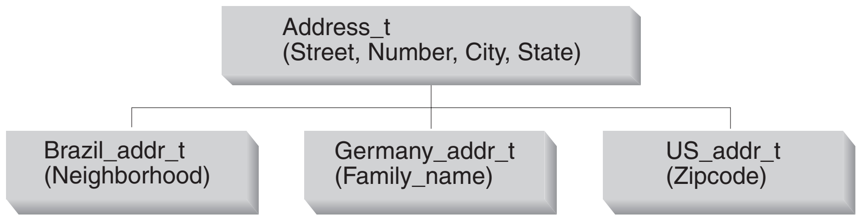


Figure 12. Structured type hierarchy for Address_t type

```
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 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 245

Related tasks:

- "Creating structured types" on page 246
- "Storing instances of structured types" on page 247
- "Inserting structured type attributes into columns" on page 278
- "Retrieving and modifying structured type values in columns" on page 281

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 254

Related tasks:

- “Creating structured types” on page 246
- “Storing structured type objects in table columns” on page 276
- “Retrieving structured type attributes” on page 282

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 245

Related tasks:

- “Creating structured types” on page 246
- “Storing structured type objects in table columns” on page 276

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:
- "Creating structured types" on page 246
- "Storing structured type objects in table columns" on page 276

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 259
- "Typed tables" on page 255

Related tasks:

- "Creating structured types" on page 246
- "Storing structured type objects in table columns" on page 276
- "Inserting structured type attributes into columns" on page 278
- "Defining and altering tables with structured type columns" on page 279
- "Defining types with structured type attributes" on page 279

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 284

Related tasks:

- "Retrieving structured type attributes" on page 282
- "Accessing the attributes of subtypes" on page 283
- "Modifying structured type attributes" on page 283
- "Returning information about a structured type" on page 284
- "Storing structured type objects in table columns" on page 276
- "Inserting structured type attributes into columns" on page 278
- "Inserting rows that contain structured type values" on page 280

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 278
- "Accessing the attributes of subtypes" on page 283
- "Modifying structured type attributes" on page 283
- "Returning information about a structured type" on page 284

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 278
- "Retrieving structured type attributes" on page 282
- "Modifying structured type attributes" on page 283
- "Returning information about a structured type" on page 284

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 278
- "Retrieving structured type attributes" on page 282
- "Accessing the attributes of subtypes" on page 283
- "Returning information about a structured type" on page 284

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 278
- "Retrieving structured type attributes" on page 282
- "Accessing the attributes of subtypes" on page 283
- "Modifying structured type attributes" on page 283

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 245
- "Transform function requirements" on page 298
- "Specification of transform groups" on page 286
- "Host language program mappings with transform functions" on page 288
- "Function transforms" on page 289
- "Recommendations for naming transform groups" on page 285

Related tasks:

- "Retrieving and modifying structured type values in columns" on page 281

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 might 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 284
- "Specification of transform groups" on page 286

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 284
- "Host language program mappings with transform functions" on page 288

Related tasks:
- "Specifying transform groups for external routines" on page 287
- "Specifying transform groups for dynamic SQL" on page 287
- "Specifying transform groups for static SQL" on page 287

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 284
- "Specification of transform groups" on page 286

Related tasks:

- "Defining behavior for structured types" on page 251
- "Specifying transform groups for dynamic SQL" on page 287
- "Specifying transform groups for static SQL" on page 287

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 284
- "Specification of transform groups" on page 286

Related tasks:

- "Specifying transform groups for external routines" on page 287
- "Specifying transform groups for static SQL" on page 287

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 284
- "Specification of transform groups" on page 286

Related tasks:

- "Specifying transform groups for external routines" on page 287
- "Specifying transform groups for dynamic SQL" on page 287

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 298
- "Transform functions and transform groups" on page 284
- "Function transforms" on page 289
- "Client transforms" on page 294

Related tasks:
- "Implementing function transforms using SQL-bodied routines" on page 291
- "Passing structured type parameters to external routines" on page 292

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 13 on page 290 shows how DB2 processes the address.

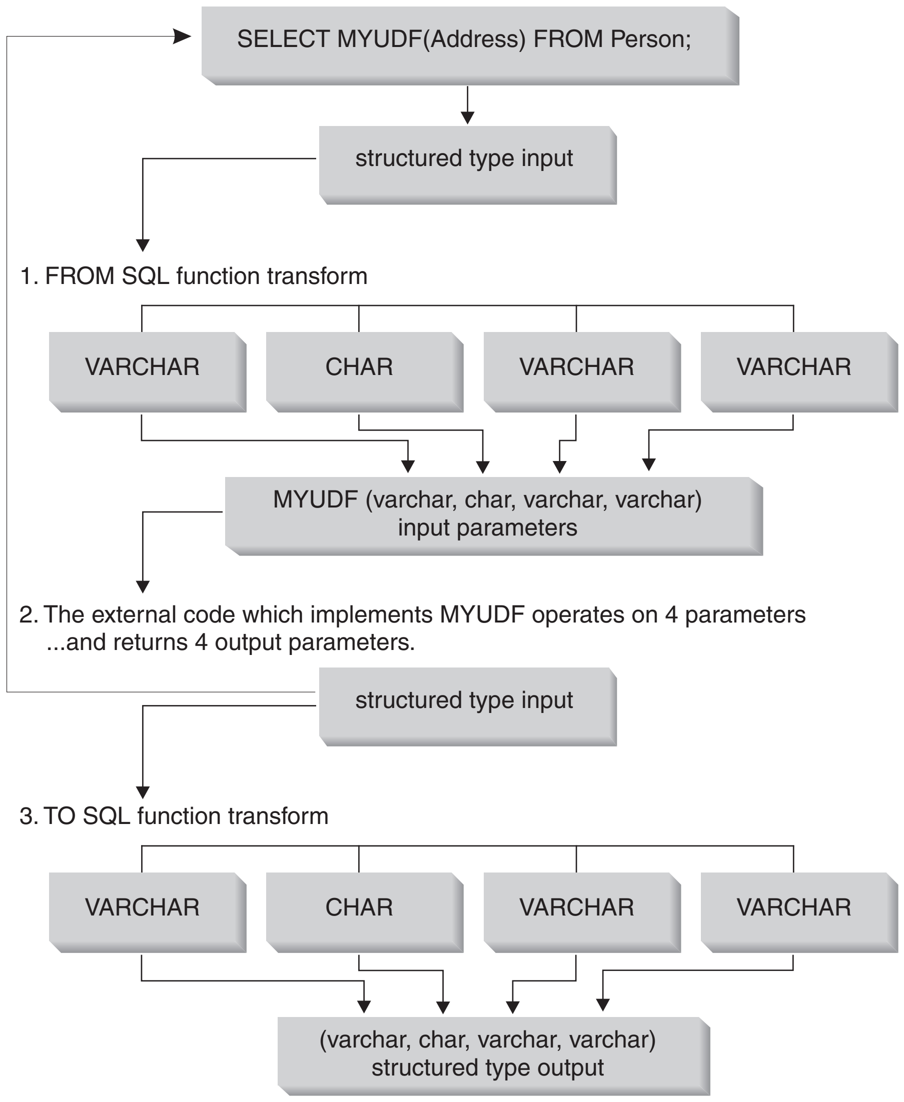

SELECT MYUDF(Address) FROM Person;
structured type input
1. FROM SQL function transform
VARCHAR
CHAR
VARCHAR
VARCHAR
MYUDF (varchar, char, varchar, varchar)
input parameters
2. The external code which implements MYUDF operates on 4 parameters
...and returns 4 output parameters.
structured type input
3. TO SQL function transform
VARCHAR
CHAR
VARCHAR
VARCHAR
(varchar, char, varchar, varchar)
structured type output

Figure 13. 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 284
- "Host language program mappings with transform functions" on page 288
- "Client transforms" on page 294

Related tasks:

- "Implementing function transforms using SQL-bodied routines" on page 291
- "Passing structured type parameters to external routines" on page 292

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 289

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 35. 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 284

- "Host language program mappings with transform functions" on page 288
- "Function transforms" on page 289
- "Client transforms" on page 294

Related tasks:

- "Implementing client transforms for binding in from a client using external UDFs" on page 296

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 14 shows the process of binding out that address to the client program.

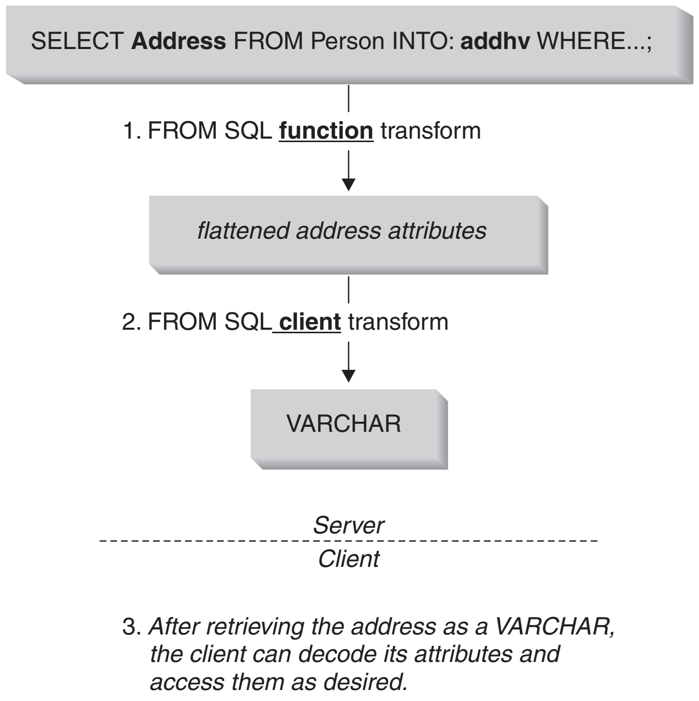

SELECT Address FROM Person INTO: addhv WHERE...;
1. FROM SQL function transform
flattened address attributes
2. FROM SQL client transform
VARCHAR
Server
Client
3. After retrieving the address as a VARCHAR, the client can decode its attributes and access them as desired.

Figure 14. 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 15 shows the reverse process of passing the address back to the database.

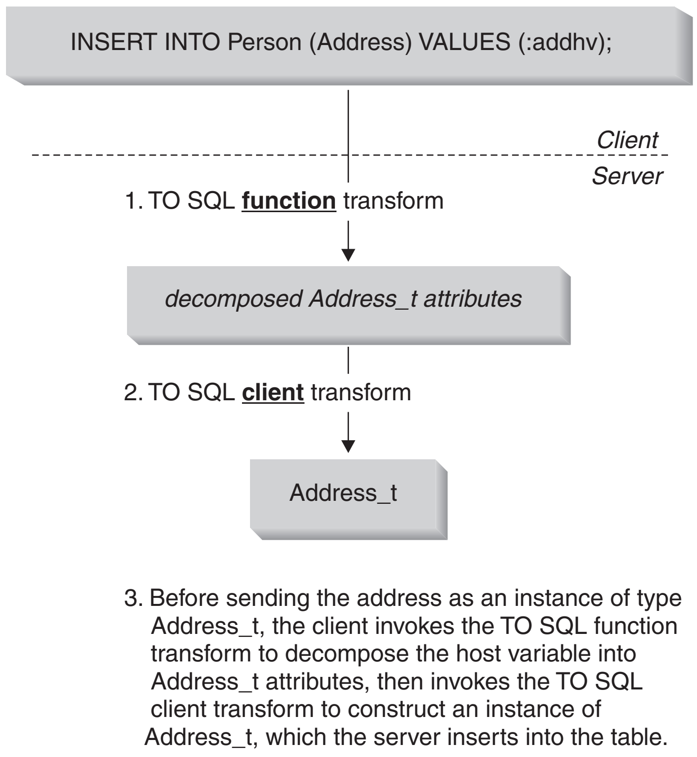

INSERT INTO Person (Address) VALUES (:addhv);
Client
Server
1. TO SQL function transform
decomposed Address_t attributes
2. TO SQL client transform
Address_t
3. Before sending the address as an instance of type Address_t, the client invokes the TO SQL function transform to decompose the host variable 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 15. 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 288
- "Function transforms" on page 289

Related tasks:

- "Implementing client transforms using external UDFs" on page 296
- "Implementing client transforms for binding in from a client using external UDFs" on page 296

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 294

Related tasks:

- "Passing structured type parameters to external routines" on page 292
- "Implementing client transforms for binding in from a client using external UDFs" on page 296

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 294

Related tasks:
- "Implementing client transforms using external UDFs" on page 296

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 operating systems 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 an operating system-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 operating system.
  - 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 an operating system-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 might have to exert considerable effort to revise your original design to correctly handle data conversion.

Related concepts:

- "Transform functions and transform groups" on page 284
- "Host language program mappings with transform functions" on page 288
- "Function transforms" on page 289

Transform function requirements

Table 36 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 36. 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 284

Related tasks:

- "Retrieving subtype data from DB2" on page 299

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 it be assumed 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. `Addr_stream` is a SQL-bodied function, defined in Step 4 on page 300, 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 299. 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 299 name the TRANSFORM GROUP that contains these transforms.

The CREATE FUNCTION statements for the transform functions are issued in Step 1 on page 299. The CREATE TRANSFORM statements that associate the transform functions with their transform groups are issued in Step 2 on page 299.

Related concepts:

- “Transform function requirements” on page 298

- “Transform functions and transform groups” on page 284

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, 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 302. 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 302 name the TRANSFORM GROUP that contains the transforms.

The SQL-bodied transform functions from Step 1 on page 302, and the associations with the transform groups from Step 2 on page 302, are named in the CREATE FUNCTION statements of Step 3 on page 302.

Related concepts:

- "Transform function requirements" on page 298
- "Transform functions and transform groups" on page 284

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 284

Related tasks:

- "Describing a structured type" on page 305

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 284

Related tasks:

- “Declaring structured type host variables” on page 305

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 an SQL trigger.

An SQL *trigger* is a named rule that is associated with a single base table. A trigger specifies actions that are to be conditionally activated upon the occurrence of a trigger event where the trigger event is a table modification (INSERT, UPDATE or DELETE) that targets a particular base table. A trigger also specifies when the trigger action is to take place; either before or after the trigger event occurs. Triggers are created and dropped with the CREATE TRIGGER and DROP TRIGGER statements. The following figure illustrates the basic logical structure of a trigger:

A trigger can be thought of as a piece of logic as follows: when an event occurs, if a prescribed condition is true, then execute an action. The event is a database operation on a table. The condition can be a condition of the state of the database or a transitional state of the table upon the event of the operation. The action can be the execution of one or more SQL statements that effect further changes to the database, the raising of an exception to prevent the modify operation from taking place, a fix-up of data modified in the event operation, or anything that can logically be contained in a procedure or function invocation. Procedures and functions can contain complex logic and can be used as sub-routines in the trigger. External user-defined functions and procedures can enable a trigger to send an e-mail or to write data to a file in the filesystem.

The following diagram shows the logical structure of a trigger:

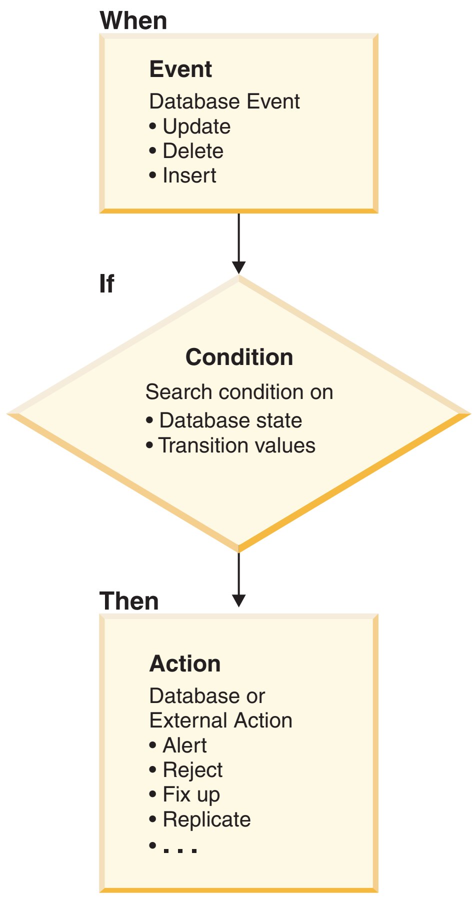


Figure 16. Classifications of routines

You can use triggers to support general forms of integrity such as business rules. For example, your business might want 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 are 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 stored procedure (serial only) or 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 mmight 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 runtime 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 310
- "Trigger granularity" on page 314

- “Trigger activation time” on page 315
- “Trigger interactions with referential constraints” on page 311
- “Trigger creation guidelines” on page 313
- “INSTEAD OF triggers” on page 311

Related tasks:
- “Creating triggers” on page 314
- “Defining business rules using triggers” on page 327

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. Connect to ’sample’ database using JDBC type 2 driver (JDBC)”
- “TbTrig.out -- HOW TO USE TRIGGERS. Connect to ’sample’ database using JDBC type 2 driver (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
  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
|     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 314
- "Trigger activation time" on page 315
- "Trigger interactions with referential constraints" on page 311
- "Triggers in application development" on page 307
- "INSTEAD OF triggers" on page 311

Related tasks:
- "Creating triggers" on page 314

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 310
- "Trigger granularity" on page 314
- "Triggers in application development" on page 307

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 FOR EACH ROW
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
  FOR EACH ROW
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
DELETE FROM EMPLOYEE AS E WHERE E.EMPNO = OLDEMP.EMPNO
```

Related concepts:

- "INSERT, UPDATE, and DELETE triggers" on page 310
- "Triggers in application development" on page 307

Related tasks:

- "Creating triggers" on page 314

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 310
- "Trigger granularity" on page 314
- "Trigger activation time" on page 315
- "Triggers in application development" on page 307

Related tasks:

- "Creating triggers" on page 314

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
 UPDATE COMPANY_STATS SET NBEMP = NBEMP+1;
```

Related concepts:

- “INSERT, UPDATE, and DELETE triggers” on page 310
- “Trigger granularity” on page 314
- “Trigger activation time” on page 315
- “Triggers in application development” on page 307
- “Trigger creation guidelines” on page 313

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. Connect to ’sample’ database using JDBC type 2 driver (JDBC)”
- “TbTrig.out -- HOW TO USE TRIGGERS. Connect to ’sample’ database using JDBC type 2 driver (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
|   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
|   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 310
- "Trigger activation time" on page 315
- "Triggers in application development" on page 307
- "Trigger creation guidelines" on page 313

Related tasks:
- "Creating triggers" on page 314

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
|   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 might 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 following diagram illustrates the execution model of before and after triggers:

|

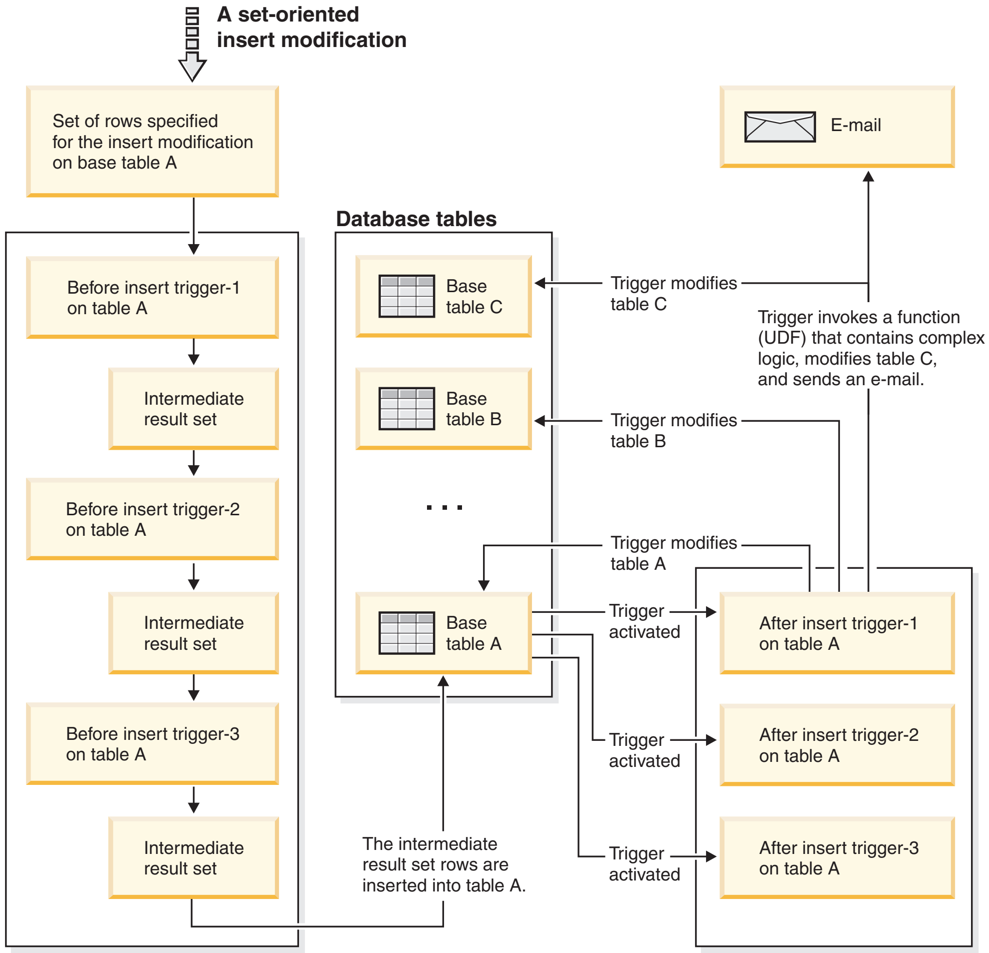

A set-oriented insert modification
Set of rows specified for the insert modification on base table A
E-mail
Database tables
Before insert trigger-1 on table A
Base table C
Trigger modifies table C
Trigger invokes a function (UDF) that contains complex logic, modifies table C, and sends an e-mail.
Intermediate result set
Base table B
Trigger modifies table B
Before insert trigger-2 on table A
Trigger modifies table A
Intermediate result set
Base table A
Trigger activated
After insert trigger-1 on table A
Before insert trigger-3 on table A
Trigger activated
After insert trigger-2 on table A
Intermediate result set
The intermediate result set rows are inserted into table A.
Trigger activated
After insert trigger-3 on table A

Figure 17. Classifications of routines

| For a given table with both before and after triggers, and a modifying event that is
| associated with these triggers, all the before triggers are activated first. The first
| activated before trigger for a given event takes operates on the set of rows targeted
| by the operation and makes any update modifications to the set that its logic
| prescribes. The output of this before trigger is accepted as input by the next
| before-trigger. When all of the before triggers that are activated by the event have
| been fired, the intermediate result set, the result of the before trigger modifications
| to the rows targeted by the trigger event operation, is applied to the base table.

Then each after trigger associated with the event is fired. The after triggers might modify the same table, another table, or perform an action external to the database.

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.

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 constraint validations. 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 can be a dynamic compound statement. However, BEFORE triggers face some restrictions; they cannot contain the following SQL statements:
- UPDATE
- DELETE
- INSERT

Related concepts:
- “INSERT, UPDATE, and DELETE triggers” on page 310
- “Trigger granularity” on page 314
- “Triggered action composed of SQL statements” on page 321
- “Triggers in application development” on page 307
- “Trigger creation guidelines” on page 313

Related tasks:

- "Creating triggers" on page 314

Transition variables

When you implement a FOR EACH ROW trigger, it might 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 can 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
  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 310
- "Trigger granularity" on page 314
- "Transition tables" on page 319

Related tasks:
- "Creating triggers" on page 314

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

Transition tables

In both FOR EACH ROW and FOR EACH STATEMENT triggers, it might 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 can 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
  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 might 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 310
- "Trigger granularity" on page 314
- "Transition variables" on page 318

Related tasks:
- "Creating triggers" on page 314

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

  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 actions qualified by conditions" on page 321
- "Triggered action composed of SQL statements" on page 321
- "Triggered action containing a procedure or function reference" on page 322

Triggered actions qualified by 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 320
- "Triggered action composed of SQL statements" on page 321
- "Triggered action containing a procedure or function reference" on page 322

Triggered action composed of 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 might 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 can be a dynamic compound statement. That is, they can 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 can contain one or more of the following:
- UPDATE SQL statement
- DELETE SQL statement
- INSERT SQL statement

Related concepts:
- "Triggered action" on page 320
- "Triggered actions qualified by conditions" on page 321
- "Triggered action containing a procedure or function reference" on page 322

Triggered action containing a procedure or function reference

Procedures and functions, including user-defined functions (UDFs), can be invoked from within the triggered action of a trigger. Procedures can be invoked using the CALL statement. Functions can be invoked within any triggered SQL statement. Invoking a routine from a trigger enables the trigger to contain complex logic. Consider the following example which shows the definition of a trigger that contains an invocation of an SQL procedure named TOTAL_SALES:

```
CREATE TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (n.c1 > 100);
BEGIN ATOMIC
   DECLARE rs INTEGER DEFAULT 0;
   CALL TOTAL_SALES(n.c1, n.c2);
   GET DIANOSTICS rs = RETURN_STATUS;
   VALUES(CASE WHEN rc < 0
               THEN RAISE_ERROR('70001',
                      'PROC CALL failed'));
END;
```

The procedure can be considered as a sub-routine to the trigger. After the SQL procedure has been invoked, the return status of the procedure is checked by executing the GET DIAGNOSTICS statement. An error is raised if the return status indicates an error occurred in the procedure.

The following is an example of a function reference within the body of a trigger. The function is referenced within the VALUES clause.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW
  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
```

The function ISSUE_SHIP_REQUEST could be an external function that sends an email to a shipping department notifying them that an order of a part is required. The function takes an expression containing transition variables as a parameter.

When a triggered action contains a procedure call with an unqualified procedure name or a triggered action SQL statement that contains a function reference with an unqualified function name, the procedure or function is resolved based on the following:

- the SQL path at the time of creation of the trigger.
- the EXECUTE privilege on the routine held by creator of the trigger.

Routines can be written in SQL, Java™, C, C++, or a .NET language. 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, a UDF called from a trigger 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
  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 320
- "Triggered actions qualified by conditions" on page 321
- "Triggered action composed of SQL statements" on page 321
- "Routine invocation" on page 193

Related tasks:
- "Invoking user-defined table functions" on page 212
- "Calling procedures from triggers or SQL routines" on page 202

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
  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
     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 307

Related tasks:
- "Extracting information from UDTs, UDFs, and LOBs with triggers" on page 325
- "Preventing operations on tables using triggers" on page 326
- "Defining business rules using triggers" on page 327
- "Defining actions using triggers" on page 328

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 a stored procedure or UDF 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 stored procedure or UDF is 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
  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 320
- "Triggered actions qualified by conditions" on page 321
- "Triggered action composed of SQL statements" on page 321
- "Triggered action containing a procedure or function reference" on page 322
- "Multiple triggers" on page 324

Related tasks:

- "Preventing operations on tables using triggers" on page 326
- "Defining business rules using triggers" on page 327
- "Defining actions using triggers" on page 328

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
  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 324
- "Triggers in application development" on page 307

Related tasks:

- "Extracting information from UDTs, UDFs, and LOBs with triggers" on page 325
- "Defining business rules using triggers" on page 327
- "Defining actions using triggers" on page 328

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
  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 324
- "Triggers in application development" on page 307

Related tasks:

- "Extracting information from UDTs, UDFs, and LOBs with triggers" on page 325
- "Preventing operations on tables using triggers" on page 326
- "Defining actions using triggers" on page 328

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
  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 324
- "Triggers in application development" on page 307

Related tasks:

- "Extracting information from UDTs, UDFs, and LOBs with triggers" on page 325
- "Preventing operations on tables using triggers" on page 326
- "Defining business rules using triggers" on page 327

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 can 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 334
- "Java routines" on page 167
- "Table function execution model for Java" on page 59

Related reference:

- "Java debug table DB2DBG.ROUTINE_DEBUG" on page 178
- "JAR file administration on the database server" on page 173
- "Supported SQL data types in DB2GENERAL routines" on page 336
- "Java classes for DB2GENERAL routines" on page 337
- "DB2GENERAL Java class: COM.IBM.db2.app.StoredProc" on page 338
- "DB2GENERAL Java class: COM.IBM.db2.app.UDF" on page 339
- "DB2GENERAL Java class: COM.IBM.db2.app.Lob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Blob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Clob" on page 343

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 can 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 can 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 can 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 333

- “Java routines” on page 167
- “Table function execution model for Java” on page 59

Related reference:

- “Java debug table DB2DBG.ROUTINE_DEBUG” on page 178
- “Supported SQL data types in DB2GENERAL routines” on page 336
- “Java classes for DB2GENERAL routines” on page 337
- “DB2GENERAL Java class: COM.IBM.db2.app.StoredProc” on page 338
- “DB2GENERAL Java class: COM.IBM.db2.app.UDF” on page 339
- “DB2GENERAL Java class: COM.IBM.db2.app.Lob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Blob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Clob” on page 343

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 37. 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
DBCLOB(*n*)[2]	COM.ibm.db2.app.Clob

Table 37. DB2 SQL Types and Java Objects (continued)

SQL Column Type	Java Data Type
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 can 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 can be processed characters-at-a-time.

If such an object is returned as an output using the `set()` method, code page conversions might be applied in order to represent the Java Unicode characters in the database code page.

Related concepts:

- “DB2GENERAL routines” on page 333
- “DB2GENERAL UDFs” on page 334
- “Java routines” on page 167
- “Table function execution model for Java” on page 59

Related reference:

- “Supported SQL data types in Java” on page 170
- “Java classes for DB2GENERAL routines” on page 337
- “DB2GENERAL Java class: COM.IBM.db2.app.StoredProc” on page 338
- “DB2GENERAL Java class: COM.IBM.db2.app.UDF” on page 339
- “DB2GENERAL Java class: COM.IBM.db2.app.Lob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Blob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Clob” on page 343

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 333
- "DB2GENERAL UDFs" on page 334
- "Java routines" on page 167

Related reference:

- "Supported SQL data types in DB2GENERAL routines" on page 336
- "DB2GENERAL Java class: COM.IBM.db2.app.StoredProc" on page 338
- "DB2GENERAL Java class: COM.IBM.db2.app.UDF" on page 339
- "DB2GENERAL Java class: COM.IBM.db2.app.Lob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Blob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Clob" on page 343

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-setting or status-setting calls 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 333
- "DB2GENERAL UDFs" on page 334
- "Java routines" on page 167

Related reference:

- "Supported SQL data types in DB2GENERAL routines" on page 336
- "Java classes for DB2GENERAL routines" on page 337
- "DB2GENERAL Java class: COM.IBM.db2.app.UDF" on page 339
- "DB2GENERAL Java class: COM.IBM.db2.app.Lob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Blob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Clob" on page 343

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 can 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 can 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 333
- “DB2GENERAL UDFs” on page 334
- “Java routines” on page 167

Related reference:

- “Supported SQL data types in DB2GENERAL routines” on page 336
- “Java classes for DB2GENERAL routines” on page 337

- “DB2GENERAL Java class: COM.IBM.db2.app.StoredProc” on page 338
- “DB2GENERAL Java class: COM.IBM.db2.app.Lob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Blob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Clob” on page 343

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 333
- “DB2GENERAL UDFs” on page 334
- “Java routines” on page 167

Related reference:

- “Supported SQL data types in DB2GENERAL routines” on page 336
- “Java classes for DB2GENERAL routines” on page 337
- “DB2GENERAL Java class: COM.IBM.db2.app.StoredProc” on page 338
- “DB2GENERAL Java class: COM.IBM.db2.app.UDF” on page 339
- “DB2GENERAL Java class: COM.IBM.db2.app.Blob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Clob” on page 343

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 can be passed back as output. The application might 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 333
- "DB2GENERAL UDFs" on page 334
- "Java routines" on page 167

Related reference:

- "Supported SQL data types in DB2GENERAL routines" on page 336
- "Java classes for DB2GENERAL routines" on page 337
- "DB2GENERAL Java class: COM.IBM.db2.app.StoredProc" on page 338
- "DB2GENERAL Java class: COM.IBM.db2.app.UDF" on page 339
- "DB2GENERAL Java class: COM.IBM.db2.app.Lob" on page 342
- "DB2GENERAL Java class: COM.IBM.db2.app.Clob" on page 343

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 can be passed back as output. The application might 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 cannot not be representede in this code page, and can cause an exception to be thrown during conversion. This can 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 333
- "DB2GENERAL UDFs" on page 334
- "Java routines" on page 167

Related reference:

- "Supported SQL data types in DB2GENERAL routines" on page 336
- "Java classes for DB2GENERAL routines" on page 337

- “DB2GENERAL Java class: COM.IBM.db2.app.StoredProc” on page 338
- “DB2GENERAL Java class: COM.IBM.db2.app.UDF” on page 339
- “DB2GENERAL Java class: COM.IBM.db2.app.Lob” on page 342
- “DB2GENERAL Java class: COM.IBM.db2.app.Blob” on page 342

Appendix B. COBOL procedures

COBOL procedures

COBOL procedures are to be written in a similar manner as COBOL subprograms.

Handling parameters in a COBOL procedure

Each parameter to be accepted or passed by a procedure must be declared in the LINKAGE SECTION. For example, this code fragment comes from a procedure that accepts two IN parameters (one CHAR(15) and one INT), and passes an OUT parameter (an INT):

```
LINKAGE SECTION.
01  IN-SPERSON    PIC X(15).
01  IN-SQTY       PIC S9(9)  USAGE COMP-5.
01  OUT-SALESSUM  PIC S9(9)  USAGE COMP-5.
```

Ensure that the COBOL data types you declare map correctly to SQL data types. For a detailed list of data type mappings between SQL and COBOL, see ″Supported SQL Data Types in COBOL″.

Each parameter must then be listed in the PROCEDURE DIVISION. The following example shows a PROCEDURE DIVISION that corresponds to the parameter definitions from the previous LINKAGE SECTION example.

```
PROCEDURE DIVISION USING IN-SPERSON
                         IN-SQTY
                         OUT-SALESSUM.
```

Exiting a COBOL procedure

To properly exit the procedure use the following commands:

```
MOVE SQLZ-HOLD-PROC TO RETURN-CODE.
GOBACK.
```

With these commands, the procedure returns correctly to the client application. This is especially important when the procedure is called by a local COBOL client application.

When building a COBOL procedure, it is strongly recommended that you use the build script written for your operating system and compiler. Build scripts for Micro Focus COBOL are found in the sqllib/samples/cobol_mf directory. Build scripts for IBM® COBOL are found in the sqllib/samples/cobol directory.

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    "NEWSALE".
DATA DIVISION.

WORKING-STORAGE SECTION.
01  INSERT-STMT.
    05  FILLER   PIC X(24) VALUE "INSERT INTO SALES (SALES".
    05  FILLER   PIC X(24) VALUE "_PERSON,SALES) VALUES ('".
    05  SPERSON  PIC X(16).
    05  FILLER   PIC X(2) VALUE "',".
    05  SQTY     PIC S9(9).
```

```
           05  FILLER   PIC X(1) VALUE ")".
           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01  INS-SMT-INF.
           05  INS-STMT.
           49  INS-LEN   PIC S9(4) USAGE COMP.
           49  INS-TEXT  PIC X(100).
       01  SALESSUM      PIC S9(9)  USAGE COMP-5.
           EXEC SQL END DECLARE SECTION END-EXEC.
           EXEC SQL INCLUDE SQLCA END-EXEC.

       LINKAGE SECTION.
       01  IN-SPERSON    PIC X(15).
       01  IN-SQTY       PIC S9(9)  USAGE COMP-5.
       01  OUT-SALESSUM  PIC S9(9)  USAGE COMP-5.

       PROCEDURE DIVISION USING IN-SPERSON
                                IN-SQTY
                                OUT-SALESSUM.
       MAINLINE.
           MOVE 0 TO SQLCODE.
           PERFORM INSERT-ROW.
           IF SQLCODE IS NOT EQUAL TO 0
              GOBACK
           END-IF.
           PERFORM SELECT-ROWS.
           PERFORM GET-SUM.
           GOBACK.
       INSERT-ROW.
           MOVE IN-SPERSON TO SPERSON.
           MOVE IN-SQTY TO SQTY.
           MOVE           INSERT-STMT TO INS-TEXT.
           MOVE LENGTH OF INSERT-STMT TO INS-LEN.
           EXEC SQL EXECUTE IMMEDIATE :INS-STMT END-EXEC.
       GET-SUM.
           EXEC SQL
              SELECT SUM(SALES) INTO :SALESSUM FROM SALES
           END-EXEC.
           MOVE SALESSUM TO OUT-SALESSUM.
       SELECT-ROWS.
           EXEC SQL
              DECLARE CUR CURSOR WITH RETURN FOR SELECT * FROM SALES
           END-EXEC.
           IF SQLCODE = 0
              EXEC SQL OPEN CUR END-EXEC
           END-IF.
```

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

```
CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                           IN SALESQTY INT,
                           OUT SALESSUM INT)
  RESULT SETS 1
  EXTERNAL NAME 'NEWSALE!NEWSALE'
  FENCED
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  MODIFIES SQL DATA
```

The preceding statement assumes that the COBOL function exists in a library called NEWSALE.

Note: When registering a COBOL procedure on Windows® operating systems, take the following precaution when identifying a stored procedure body in the

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

```
CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                           IN SALESQTY INT,
                           OUT SALESSUM INT)
  RESULT SETS 1
  EXTERNAL NAME 'NEWSALE!NEWSALE'
  FENCED
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  MODIFIES SQL DATA
  EXTERNAL NAME 'd:\mylib\NEWSALE.dll'
```

Related concepts:

- "Procedures" on page 11
- "Embedded SQL Statements in COBOL" in the *Application Development Guide: Programming Client Applications*
- "Host Variables in COBOL" in the *Application Development Guide: Programming Client Applications*

Related tasks:

- "Building IBM COBOL routines on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX Micro Focus COBOL routines" 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*

Related reference:

- "Supported SQL Data Types in COBOL" on page 347
- "CREATE PROCEDURE (External) statement" in the *SQL Reference, Volume 2*
- "Syntax for passing arguments to routines written in C/C++, OLE, or COBOL" on page 89

Supported SQL Data Types in COBOL

Certain predefined COBOL data types correspond to column types. Only these COBOL data types can be declared as host variables.

The following table shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

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

Table 38. SQL Data Types Mapped to COBOL Declarations

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
SMALLINT (500 or 501)	01 name PIC S9(4) COMP-5.	16-bit signed integer
INTEGER (496 or 497)	01 name PIC S9(9) COMP-5.	32-bit signed integer
BIGINT (492 or 493)	01 name PIC S9(18) COMP-5.	64-bit signed integer
DECIMAL(*p*,*s*) (484 or 485)	01 name PIC S9(*m*)V9(*n*) COMP-3.	Packed decimal
REAL[2] (480 or 481)	01 name USAGE IS COMP-1.	Single-precision floating point
DOUBLE[3] (480 or 481)	01 name USAGE IS COMP-2.	Double-precision floating point
CHAR(*n*) (452 or 453)	01 name PIC X(*n*).	Fixed-length character string
VARCHAR(*n*) (448 or 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=32 672	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*n*<=32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (920 or 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	Identifies BLOB entities residing on the server
BLOB file reference variable[4] (916 or 917)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
DATE (384 or 385)	01 identifier PIC X(10).	10-byte character string
TIME (388 or 389)	01 identifier PIC X(8).	8-byte character string
TIMESTAMP (392 or 393)	01 identifier PIC X(26).	26-byte character string
Note: The following data types are only available in the DBCS environment.		
GRAPHIC(*n*) (468 or 469)	01 name PIC G(*n*) DISPLAY-1.	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	Variable length double-byte character string with 2-byte string length indicator

Table 38. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
LONG VARGRAPHIC (472 or 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 16 337<=*n*<=16 350	Variable length double-byte character string with 2-byte string length indicator
DBCLOB(*n*) (412 or 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n). 1<=*n*<=1 073 741 823	Large object variable-length double-byte character string with 4-byte string length indicator
DBCLOB locator variable[4] (968 or 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (924 or 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following are additional rules for supported COBOL data types:

- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. See the following sample:

  ```
  01 identifier PIC S9(m)V9(n) COMP-3
  ```

  – Use V to denote the decimal point.
  – Values for *n* and *m* must be greater than or equal to 1.
  – The value for $n + m$ cannot exceed 31.
  – The value for *s* equals the value for *n*.
  – The value for *p* equals the value for $n + m$.
  – The repetition factors *(n)* and *(m)* are optional. The following examples are all valid:

  ```
  01 identifier PIC S9(3)V COMP-3
  01 identifier PIC SV9(3) COMP-3
  01 identifier PIC S9V COMP-3
  01 identifier PIC SV9 COMP-3
  ```

  – PACKED-DECIMAL can be used instead of COMP-3.
- Arrays are *not* supported by the COBOL precompiler.

Related concepts:

- "SQL Declare Section with Host Variables for COBOL" in the *Application Development Guide: Programming Client Applications*

Appendix C. DB2 Universal Database technical information

DB2 documentation and help

DB2® technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- Downloadable PDF files, PDF files on CD, and printed books
  - Guides
  - Reference manuals
- Command line help
  - Command help
  - Message help
  - SQL state help
- Installed source code
  - Sample programs

You can access additional DB2 Universal Database™ technical information such as technotes, white papers, and Redbooks™ online at ibm.com®. Access the DB2 Information Management software library site at www.ibm.com/software/data/pubs/.

DB2 documentation updates

IBM® may periodically make documentation FixPaks and other documentation updates to the DB2 Information Center available. If you access the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/, you will always be viewing the most up-to-date information. If you have installed the DB2 Information Center locally, then you need to install any updates manually before you can view them. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* when new information becomes available.

The Information Center is updated more frequently than either the PDF or the hardcopy books. To get the most current DB2 technical information, install the documentation updates as they become available or go to the DB2 Information Center at the www.ibm.com site.

Related concepts:

- "CLI sample programs" in the *CLI Guide and Reference, Volume 1*
- "Java sample programs" in the *Application Development Guide: Building and Running Applications*
- "DB2 Information Center" on page 352

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 369

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Invoking message help from the command line processor" on page 370
- "Invoking command help from the command line processor" on page 370
- "Invoking SQL state help from the command line processor" on page 371

Related reference:

- "DB2 PDF and printed documentation" on page 363

DB2 Information Center

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 family products, including DB2 Universal Database™, DB2 Connect™, DB2 Information Integrator and DB2 Query Patroller™. The DB2 Information Center also contains information for major DB2 features and components including replication, data warehousing, and the DB2 extenders.

The DB2 Information Center has the following features if you view it in Mozilla 1.0 or later or Microsoft® Internet Explorer 5.5 or later. Some features require you to enable support for JavaScript™:

Flexible installation options
You can choose to view the DB2 documentation using the option that best meets your needs:

- To effortlessly ensure that your documentation is always up to date, you can access all of your documentation directly from the DB2 Information Center hosted on the IBM® Web site at http://publib.boulder.ibm.com/infocenter/db2help/
- To minimize your update efforts and keep your network traffic within your intranet, you can install the DB2 documentation on a single server on your intranet
- To maximize your flexibility and reduce your dependence on network connections, you can install the DB2 documentation on your own computer

Search
| You can search all of the topics in the DB2 Information Center by entering
| a search term in the **Search** text field. You can retrieve exact matches by
| enclosing terms in quotation marks, and you can refine your search with
| wildcard operators (*, ?) and Boolean operators (AND, NOT, OR).

Task-oriented table of contents
| You can locate topics in the DB2 documentation from a single table of
| contents. The table of contents is organized primarily by the kind of tasks
| you may want to perform, but also includes entries for product overviews,
| goals, reference information, an index, and a glossary.

| - Product overviews describe the relationship between the available
| products in the DB2 family, the features offered by each of those
| products, and up to date release information for each of these products.
| - Goal categories such as installing, administering, and developing include
| topics that enable you to quickly complete tasks and develop a deeper
| understanding of the background information for completing those
| tasks.

- Reference topics provide detailed information about a subject, including statement and command syntax, message help, and configuration parameters.

Show current topic in table of contents
You can show where the current topic fits into the table of contents by clicking the **Refresh / Show Current Topic** button in the table of contents frame or by clicking the **Show in Table of Contents** button in the content frame. This feature is helpful if you have followed several links to related topics in several files or arrived at a topic from search results.

Index You can access all of the documentation from the index. The index is organized in alphabetical order by index term.

Glossary
You can use the glossary to look up definitions of terms used in the DB2 documentation. The glossary is organized in alphabetical order by glossary term.

Integrated localized information
The DB2 Information Center displays information in the preferred language set in your browser preferences. If a topic is not available in your preferred language, the DB2 Information Center displays the English version of that topic.

For iSeries™ technical information, refer to the IBM eServer™ iSeries information center at www.ibm.com/eserver/iseries/infocenter/.

Related concepts:
- "DB2 Information Center installation scenarios" on page 353

Related tasks:
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Displaying topics in your preferred language in the DB2 Information Center" on page 362
- "Invoking the DB2 Information Center" on page 360
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 356
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 358

DB2 Information Center installation scenarios

Different working environments can pose different requirements for how to access DB2® information. The DB2 Information Center can be accessed on the IBM® Web site, on a server on your organization's network, or on a version installed on your computer. In all three cases, the documentation is contained in the DB2 Information Center, which is an architected web of topic-based information that you view with a browser. By default, DB2 products access the DB2 Information Center on the IBM Web site. However, if you want to access the DB2 Information Center on an intranet server or on your own computer, you must install the DB2 Information Center using the DB2 Information Center CD found in your product Media Pack. Refer to the summary of options for accessing DB2 documentation which follows, along with the three installation scenarios, to help determine which method of accessing the DB2 Information Center works best for you and your work environment, and what installation issues you might need to consider.

Summary of options for accessing DB2 documentation:

The following table provides recommendations on which options are possible in your work environment for accessing the DB2 product documentation in the DB2 Information Center.

Internet access	Intranet access	Recommendation
Yes	Yes	Access the DB2 Information Center on the IBM Web site, or access the DB2 Information Center installed on an intranet server.
Yes	No	Access the DB2 Information Center on the IBM Web site.
No	Yes	Access the DB2 Information Center installed on an intranet server.
No	No	Access the DB2 Information Center on a local computer.

Scenario: Accessing the DB2 Information Center on your computer:

Tsu-Chen owns a factory in a small town that does not have a local ISP to provide him with Internet access. He purchased DB2 Universal Database™ to manage his inventory, his product orders, his banking account information, and his business expenses. Never having used a DB2 product before, Tsu-Chen needs to learn how to do so from the DB2 product documentation.

After installing DB2 Universal Database on his computer using the typical installation option, Tsu-Chen tries to access the DB2 documentation. However, his browser gives him an error message that the page he tried to open cannot be found. Tsu-Chen checks the installation manual for his DB2 product and discovers that he has to install the DB2 Information Center if he wants to access DB2 documentation on his computer. He finds the *DB2 Information Center CD* in the media pack and installs it.

From the application launcher for his operating system, Tsu-Chen now has access to the DB2 Information Center and can learn how to use his DB2 product to increase the success of his business.

Scenario: Accessing the DB2 Information Center on the IBM Web site:

Colin is an information technology consultant with a training firm. He specializes in database technology and SQL and gives seminars on these subjects to businesses all over North America using DB2 Universal Database. Part of Colin's seminars includes using DB2 documentation as a teaching tool. For example, while teaching courses on SQL, Colin uses the DB2 documentation on SQL as a way to teach basic and advanced syntax for database queries.

Most of the businesses at which Colin teaches have Internet access. This situation influenced Colin's decision to configure his mobile computer to access the DB2 Information Center on the IBM Web site when he installed the latest version of DB2 Universal Database. This configuration allows Colin to have online access to the latest DB2 documentation during his seminars.

However, sometimes while travelling Colin does not have Internet access. This posed a problem for him, especially when he needed to access to DB2 documentation to prepare for seminars. To avoid situations like this, Colin installed a copy of the DB2 Information Center on his mobile computer.

Colin enjoys the flexibility of always having a copy of DB2 documentation at his disposal. Using the **db2set** command, he can easily configure the registry variables on his mobile computer to access the DB2 Information Center on either the IBM Web site, or his mobile computer, depending on his situation.

Scenario: Accessing the DB2 Information Center on an intranet server:

Eva works as a senior database administrator for a life insurance company. Her administration responsibilities include installing and configuring the latest version of DB2 Universal Database on the company's UNIX® database servers. Her company recently informed its employees that, for security reasons, it would not provide them with Internet access at work. Because her company has a networked environment, Eva decides to install a copy of the DB2 Information Center on an intranet server so that all employees in the company who use the company's data warehouse on a regular basis (sales representatives, sales managers, and business analysts) have access to DB2 documentation.

Eva instructs her database team to install the latest version of DB2 Universal Database on all of the employee's computers using a response file, to ensure that each computer is configured to access the DB2 Information Center using the host name and the port number of the intranet server.

However, through a misunderstanding Migual, a junior database administrator on Eva's team, installs a copy of the DB2 Information Center on several of the employee computers, rather than configuring DB2 Universal Database to access the DB2 Information Center on the intranet server. To correct this situation Eva tells Migual to use the **db2set** command to change the DB2 Information Center registry variables (DB2_DOCHOST for the host name, and DB2_DOCPORT for the port number) on each of these computers. Now all of the appropriate computers on the network have access to the DB2 Information Center, and employees can find answers to their DB2 questions in the DB2 documentation.

Related concepts:
- "DB2 Information Center" on page 352

Related tasks:
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 356
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 358

Related reference:
- "db2set - DB2 Profile Registry Command" in the *Command Reference*

Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a UNIX operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on UNIX computers.

- **Hardware requirements**

  You require one of the following processors:
  - PowerPC (AIX)
  - HP 9000 (HP-UX)
  - Intel 32–bit (Linux)
  - Solaris UltraSPARC computers (Solaris Operating Environment)
- **Operating system requirements**

  You require one of the following operating systems:
  - IBM AIX 5.1 (on PowerPC)
  - HP-UX 11i (on HP 9000)
  - Red Hat Linux 8.0 (on Intel 32–bit)
  - SuSE Linux 8.1 (on Intel 32–bit)
  - Sun Solaris Version 8 (on Solaris Operating Environment UltraSPARC computers)

  Note: The DB2 Information Center runs on a subset of the UNIX operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center from the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**
  - The following browser is supported:
    - Mozilla Version 1.0 or greater
- The DB2 Setup wizard is a graphical installer. You must have an implementation of the X Window System software capable of rendering a graphical user interface for the DB2 Setup wizard to run on your computer. Before you can run the DB2 Setup wizard you must ensure that you have properly exported your display. For example, enter the following command at the command prompt:

  ```
  export DISPLAY=9.26.163.144:0.
  ```
- **Communication requirements**
  - TCP/IP

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system.
2. Insert and mount the DB2 Information Center product CD on your system.
3. Change to the directory where the CD is mounted by entering the following command:

   ```
   cd /cd
   ```

   where */cd* represents the mount point of the CD.
4. Enter the **./db2setup** command to start the DB2 Setup wizard.
5. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
6. On the **Select the product you would like to install** page, click **Next**.
7. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
8. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
9. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
10. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
11. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
12. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can also install the DB2 Information Center using a response file.

The installation logs db2setup.his, db2setup.log, and db2setup.err are located, by default, in the /tmp directory.

The db2setup.log file captures all DB2 product installation information, including errors. The db2setup.his file records all DB2 product installations on your computer. DB2 appends the db2setup.log file to the db2setup.his file. The db2setup.err file captures any error output that is returned by Java, for example, exceptions and trap information.

When the installation is complete, the DB2 Information Center will be installed in one of the following directories, depending upon your UNIX operating system:
- AIX: /usr/opt/db2_08_01
- HP-UX: /opt/IBM/db2/V8.1
- Linux: /opt/IBM/db2/V8.1
- Solaris Operating Environment: /opt/IBM/db2/V8.1

Related concepts:
- "DB2 Information Center" on page 352
- "DB2 Information Center installation scenarios" on page 353

Related tasks:

- "Installing DB2 using a response file (UNIX)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Displaying topics in your preferred language in the DB2 Information Center" on page 362
- "Invoking the DB2 Information Center" on page 360
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 358

Installing the DB2 Information Center using the DB2 Setup wizard (Windows)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the DB2 documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a Windows operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on Windows.

- **Hardware requirements**

  You require one of the following processors:
  - 32-bit computers: a Pentium or Pentium compatible CPU
- **Operating system requirements**

  You require one of the following operating systems:
  - Windows 2000
  - Windows XP

  Note: The DB2 Information Center runs on a subset of the Windows operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center on the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**
  - The following browsers are supported:
    - Mozilla 1.0 or greater
    - Internet Explorer Version 5.5 or 6.0 (Version 6.0 for Windows XP)
- **Communication requirements**
  - TCP/IP

Restrictions:

- You require an account with administrative privileges to install the DB2 Information Center.

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system with the account that you have defined for the DB2 Information Center installation.
2. Insert the CD into the drive. If enabled, the auto-run feature starts the IBM DB2 Setup Launchpad.
3. The DB2 Setup wizard determines the system language and launches the setup program for that language. If you want to run the setup program in a language other than English, or the setup program fails to auto-start, you can start the DB2 Setup wizard manually.

   To start the DB2 Setup wizard manually:

   a. Click **Start** and select **Run**.
   b. In the **Open** field, type the following command:
      ```
      x:\setup.exe /i 2-letter language identifier
      ```
      where *x:* represents your CD drive, and *2-letter language identifier* represents the language in which the setup program will be run.
   c. Click **OK**.
4. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
5. On the **Select the product you would like to install** page, click **Next**.
6. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
7. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
8. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
9. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
10. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
11. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can install the DB2 Information Center using a response file. You can also use the **db2rspgn** command to generate a response file based on an existing installation.

For information on errors encountered during installation, see the db2.log and db2wi.log files located in the 'My Documents'\DB2LOG\ directory. The location of the 'My Documents' directory will depend on the settings on your computer.

The db2wi.log file captures the most recent DB2 installation information. The db2.log captures the history of DB2 product installations.

Related concepts:

- "DB2 Information Center" on page 352
- "DB2 Information Center installation scenarios" on page 353

Related tasks:

- "Installing a DB2 product using a response file (Windows)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Displaying topics in your preferred language in the DB2 Information Center" on page 362
- "Invoking the DB2 Information Center" on page 360
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 356

Related reference:

- "db2rspgn - Response File Generator Command (Windows)" in the *Command Reference*

Invoking the DB2 Information Center

The DB2 Information Center gives you access to all of the information that you need to use DB2 products for Linux, UNIX, and Windows operating systems such as DB2 Universal Database, DB2 Connect, DB2 Information Integrator, and DB2 Query Patroller.

You can invoke the DB2 Information Center from one of the following places:

- Computers on which a DB2 UDB client or server is installed
- An intranet server or local computer on which the DB2 Information Center installed
- The IBM Web site

Prerequisites:

Before you invoke the DB2 Information Center:

- *Optional*: Configure your browser to display topics in your preferred language
- *Optional*: Configure your DB2 client to use the DB2 Information Center installed on your computer or intranet server

Procedure:

To invoke the DB2 Information Center on a computer on which a DB2 UDB client or server is installed:

- From the Start Menu (Windows operating system): Click **Start** → **Programs** → **IBM DB2** → **Information** → **Information Center**.
- From the command line prompt:
  - For Linux and UNIX operating systems, issue the **db2icdocs** command.
  - For the Windows operating system, issue the **db2icdocs.exe** command.

To open the DB2 Information Center installed on an intranet server or local computer in a Web browser:

- Open the Web page at http://<host-name>:<port-number>/, where <host-name> represents the host name and <port-number> represents the port number on which the DB2 Information Center is available.

To open the DB2 Information Center on the IBM Web site in a Web browser:
- Open the Web page at publib.boulder.ibm.com/infocenter/db2help/.

Related concepts:
- "DB2 Information Center" on page 352

Related tasks:
- "Displaying topics in your preferred language in the DB2 Information Center" on page 362
- "Invoking contextual help from a DB2 tool" on page 369
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 361
- "Invoking message help from the command line processor" on page 370
- "Invoking command help from the command line processor" on page 370
- "Invoking SQL state help from the command line processor" on page 371

Updating the DB2 Information Center installed on your computer or intranet server

The DB2 Information Center available from http://publib.boulder.ibm.com/infocenter/db2help/ will be periodically updated with new or changed documentation. IBM may also make DB2 Information Center updates available to download and install on your computer or intranet server. Updating the DB2 Information Center does not update DB2 client or server products.

Prerequisites:

You must have access to a computer that is connected to the Internet.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:
1. Open the DB2 Information Center hosted on the IBM Web site at: http://publib.boulder.ibm.com/infocenter/db2help/
2. In the Downloads section of the welcome page under the Service and Support heading, click the **DB2 Universal Database documentation** link.
3. Determine if the version of your DB2 Information Center is out of date by comparing the latest refreshed documentation image level to the documentation level you have installed. The documentation level you have installed is listed on the DB2 Information Center welcome page.
4. If a more recent version of the DB2 Information Center is available, download the latest refreshed *DB2 Information Center* image applicable to your operating system.
5. To install the refreshed *DB2 Information Center* image, follow the instructions provided on the Web page.

Related concepts:

- "DB2 Information Center installation scenarios" on page 353

Related tasks:

- "Invoking the DB2 Information Center" on page 360
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 356
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 358

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

Procedure:

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in the Mozilla browser:

1. In Mozilla, select the **Edit** —> **Preferences** —> **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

Related concepts:

- "DB2 Information Center" on page 352

DB2 PDF and printed documentation

The following tables provide official book names, form numbers, and PDF file names. To order hardcopy books, you must know the official book name. To print a PDF file, you must know the PDF file name.

The DB2 documentation 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, or to print or view the PDF 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

Core DB2 information

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

Table 39. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x81
IBM DB2 Universal Database Glossary	No form number	db2t0x81
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840, not available in hardcopy	db2m1x81
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841, not available in hardcopy	db2m2x81
IBM DB2 Universal Database What's New	SC09-4848	db2q0x81

Administration information

The information in these books covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

Table 40. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x81

Table 40. Administration information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x81
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x81
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x81
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx81
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax81
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx81
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x81
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x81
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x81

Application development information

The information in these books is of special interest to application developers or programmers working with DB2 Universal Database (DB2 UDB). You will find information about supported languages and compilers, as well as the documentation required to access DB2 UDB using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLJ, and CLI. If you are using the DB2 Information Center, you can also access HTML versions of the source code for the sample programs.

Table 41. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx81
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x81
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x81
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x81

Table 41. Application development information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x81
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx81
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx81

Business intelligence information

The information in these books describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

Table 42. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Standard Edition Information Catalog Center Administration Guide	SC27-1125	db2dix81
IBM DB2 Warehouse Manager Standard Edition Installation Guide	GC27-1122	db2idx81
IBM DB2 Warehouse Manager Standard Edition Managing ETI Solution Conversion Programs with DB2 Warehouse Manager	SC18-7727	iwhe1mstx80

DB2 Connect information

The information in this category describes how to access data on mainframe and midrange servers using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

Table 43. DB2 Connect information

Name	Form number	PDF file name
IBM Connectivity Supplement	No form number	db2h1x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x81
IBM DB2 Connect User's Guide	SC09-4835	db2c0x81

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

Table 44. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832, not available in hardcopy	db2itx81
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx81
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x81
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837, not available in hardcopy	db2iyx81
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x81

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

Table 45. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux81
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax81
Information Catalog Center Tutorial	No form number	db2aix81
Video Central for e-business Tutorial	No form number	db2twx81
Visual Explain Tutorial	No form number	db2tvx81

Optional component information

The information in this category describes how to work with optional DB2 components.

Table 46. Optional component information

Name	Form number	PDF file name
IBM DB2 Cube Views Guide and Reference	SC18–7298	db2aax81
IBM DB2 Query Patroller Guide: Installation, Administration and Usage Guide	GC09–7658	db2dwx81
IBM DB2 Spatial Extender and Geodetic Extender User's Guide and Reference	SC27-1226	db2sbx81

Table 46. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x82
DB2 Net Search Extender Administration and User's 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. The release notes also provide summaries of the documentation updates incorporated in each release, update, and FixPak.

Table 47. Release notes

Name	Form number	PDF file name
DB2 Release Notes	See note.	See note.
DB2 Installation Notes	Available on product CD-ROM only.	Not available.

Note: The Release Notes are available in:

- XHTML and Text format, on the product CDs
- PDF format, on the PDF Documentation CD

In addition the portions of the Release Notes that discuss *Known Problems and Workarounds* and *Incompatibilities Between Releases* also appear in the DB2 Information Center.

To view the Release Notes in text format 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:

- For AIX operating systems: /usr/opt/db2_08_01
- For all other UNIX-based operating systems: /opt/IBM/db2/V8.1

Related concepts:

- "DB2 documentation and help" on page 351

Related tasks:

- "Printing DB2 books from PDF files" on page 368
- "Ordering printed DB2 books" on page 368
- "Invoking contextual help from a DB2 tool" on page 369

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 installed. If you need to install 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. Open index.htm. The file opens in a browser window.
3. Click on the title of the PDF you want to see. The PDF will open in Acrobat Reader.
4. Select **File** → **Print** to print any portions of the book that you want.

Related concepts:
- "DB2 Information Center" on page 352

Related tasks:
- "Mounting the CD-ROM (AIX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (HP-UX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (Linux)" in the *Quick Beginnings for DB2 Servers*
- "Ordering printed DB2 books" on page 368
- "Mounting the CD-ROM (Solaris Operating Environment)" in the *Quick Beginnings for DB2 Servers*

Related reference:
- "DB2 PDF and printed documentation" on page 363

Ordering printed DB2 books

If you prefer to use hardcopy books, you can order them in one of three ways.

Procedure:

Printed books can be ordered in some countries or regions. Check the IBM Publications website for your country or region to see if this service is available in your country or region. When the publications are available for ordering, you can:
- 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/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.

- Visit the IBM Publications Center at http://www.ibm.com/shop/publications/order. The ability to order books from the IBM Publications Center may not be available in all countries.

At the time the DB2 product becomes available, the printed books are the same as those that are available in PDF format on the *DB2 PDF Documentation CD*. Content in the printed books that appears in the *DB2 Information Center CD* is also the same. However, there is some additional content available in DB2 Information Center CD that does not appear anywhere in the PDF books (for example, SQL Administration routines and HTML samples). Not all books available on the DB2 PDF Documentation CD are available for ordering in hardcopy.

Note: The DB2 Information Center is updated more frequently than either the PDF or the hardcopy books; install documentation updates as they become available or refer to the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/ to get the most current information.

Related tasks:

- "Printing DB2 books from PDF files" on page 368

Related reference:

- "DB2 PDF and printed documentation" on page 363

Invoking contextual help from a DB2 tool

Contextual help provides information about the tasks or controls that are associated with a particular window, notebook, wizard, or advisor. Contextual help is available from DB2 administration and development tools that have graphical user interfaces. There are two types of contextual help:

- Help accessed through the **Help** button that is located on each window or notebook
- Infopops, which are pop-up information windows displayed when the mouse cursor is placed over a field or control, or when a field or control is selected in a window, notebook, wizard, or advisor and F1 is pressed.

The **Help** button gives you access to overview, prerequisite, and task information. The infopops describe the individual fields and controls.

Procedure:

To invoke contextual help:

- For window and notebook help, start one of the DB2 tools, then open any window or notebook. Click the **Help** button at the bottom right corner of the window or notebook to invoke the contextual help.

  You can also access the contextual help from the **Help** menu item at the top of each of the DB2 tools centers.

  Within wizards and advisors, click on the Task Overview link on the first page to view contextual help.
- For infopop help about individual controls on a window or notebook, click the control, then click **F1**. Pop-up information containing details about the control is displayed in a yellow window.

Note: To display infopops simply by holding the mouse cursor over a field or control, select the **Automatically display infopops** check box on the **Documentation** page of the Tool Settings notebook.

Similar to infopops, diagnosis pop-up information is another form of context-sensitive help; they contain data entry rules. Diagnosis pop-up information is displayed in a purple window that appears when data that is not valid or that is insufficient is entered. Diagnosis pop-up information can appear for:

– Compulsory fields.
– Fields whose data follows a precise format, such as a date field.

Related tasks:

- "Invoking the DB2 Information Center" on page 360
- "Invoking message help from the command line processor" on page 370
- "Invoking command help from the command line processor" on page 370
- "Invoking SQL state help from the command line processor" on page 371
- "How to use the DB2 UDB help: Common GUI help"
- "Setting up access to DB2 contextual help and documentation: Common GUI help"

Invoking message help from the command line processor

Message help describes the cause of a message and describes any action you should take in response to the error.

Procedure:

To invoke message help, open the command line processor and enter:

```
   ? XXXnnnnn
```

where *XXXnnnnn* represents a valid message identifier.

For example, `? SQL30081` displays help about the SQL30081 message.

Related concepts:

- "Introduction to messages" in the *Message Reference Volume 1*

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking command help from the command line processor

Command help explains the syntax of commands in the command line processor.

Procedure:

To invoke command help, open the command line processor and enter:

```
  ? command
```

where *command* represents a keyword or the entire command.

For example, `? catalog` displays help for all of the CATALOG commands, while `? catalog database` displays help only for the CATALOG DATABASE command.

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 369
- "Invoking the DB2 Information Center" on page 360
- "Invoking message help from the command line processor" on page 370
- "Invoking SQL state help from the command line processor" on page 371

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking SQL state help from the command line processor

DB2 Univerrsal Database returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure:

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Related tasks:

- "Invoking the DB2 Information Center" on page 360
- "Invoking message help from the command line processor" on page 370
- "Invoking command help from the command line processor" on page 370

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:

You can view the XHTML versions of the tutorials from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some tutorial lessons use sample data or code. See each tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

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.

Information Catalog Center Tutorial
: Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Visual Explain Tutorial
: Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2® products.

DB2 documentation
: Troubleshooting information can be found throughout the DB2 Information Center, as well as throughout the PDF books that make up the DB2 library. You can refer to the ″Support and troubleshooting″ branch of the DB2 Information Center navigation tree (in the left pane of your browser window) to see a complete listing of the DB2 troubleshooting documentation.

DB2 Technical Support Web site
: Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs), FixPaks and the latest listing of internal DB2 error codes, and other resources. You can search through this knowledge base to find possible solutions to your problems.

  Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/winos2unix/support

DB2 Problem Determination Tutorial Series
: Refer to the DB2 Problem Determination Tutorial Series Web site to find information on how to quickly identify and resolve problems you might encounter while working with DB2 products. One tutorial introduces you to the DB2 problem determination facilities and tools available, and helps you decide when to use them. Other tutorials deal with related topics, such as ″Database Engine Problem Determination″, ″Performance Problem Determination″, and ″Application Problem Determination″.

  See the full set of DB2 problem determination tutorials on the DB2 Technical Support site at http://www.ibm.com/software/data/support/pdm/db2tutorials.html

Related concepts:

- "DB2 Information Center" on page 352
- "Introduction to problem determination - DB2 Technical Support tutorial" in the *Troubleshooting Guide*

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. The following list specifies the major accessibility features in DB2® Version 8 products:

- All DB2 functionality is available using the keyboard for navigation instead of the mouse. For more information, see "Keyboard input and navigation."
- You can customize the size and color of the fonts on DB2 user interfaces. For more information, see "Accessible display."
- DB2 products support accessibility applications that use the Java™ Accessibility API. For more information, see "Compatibility with assistive technologies" on page 374.
- DB2 documentation is provided in an accessible format. For more information, see "Accessible documentation" on page 374.

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 operations that can also be done using a mouse. Standard operating system keystrokes are used for standard operating system operations.

For more information about using keys or key combinations to perform operations, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard navigation

You can navigate the DB2 tools user interface using keys or key combinations.

For more information about using keys or key combinations to navigate the DB2 Tools, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard focus

In UNIX® operating systems, the area of the active window where your keystrokes will have an effect is highlighted.

Accessible display

The DB2 tools have features that improve accessibility for users with low vision or other visual impairments. These accessibility enhancements include support for customizable font properties.

Font settings

You can select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

For more information about specifying font settings, see Changing the fonts for menus and text: Common GUI help.

Non-dependence on color

You do not need to distinguish between colors in order to use any of the functions in this product.

Compatibility with assistive technologies

The DB2 tools interfaces support the Java Accessibility API, which enables you to use screen readers and other assistive technologies with DB2 products.

Accessible documentation

Documentation for DB2 is provided in XHTML 1.0 format, which is viewable in most Web browsers. XHTML 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.

Syntax diagrams are provided in dotted decimal format. This format is available only if you are accessing the online documentation using a screen-reader.

Related concepts:

- "Dotted decimal syntax diagrams" on page 374

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the

LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.
- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Notes:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once

and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Related concepts:

- "Accessibility" on page 373

Related tasks:

- "Contents : Common help"

Related reference:

- "How to read the syntax diagrams" in the *SQL Reference, Volume 2*

Common Criteria certification of DB2 Universal Database products

DB2 Universal Database is being evaluated for certification under the Common Criteria at evaluation assurance level 4 (EAL4). For more information about Common Criteria, see the Common Criteria web site at: http://niap.nist.gov/cc-scheme/.

Appendix D. 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 improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

ACF/VTAM	iSeries
AISPO	LAN Distance
AIX	MVS
AIXwindows	MVS/ESA
AnyNet	MVS/XA
APPN	Net.Data
AS/400	NetView
BookManager	OS/390
C Set++	OS/400
C/370	PowerPC
CICS	pSeries
Database 2	QBIC
DataHub	QMF
DataJoiner	RACF
DataPropagator	RISC System/6000
DataRefresher	RS/6000
DB2	S/370
DB2 Connect	SP
DB2 Extenders	SQL/400
DB2 OLAP Server	SQL/DS
DB2 Information Integrator	System/370
DB2 Query Patroller	System/390
DB2 Universal Database	SystemView
Distributed Relational Database Architecture	Tivoli
DRDA	VisualAge
eServer	VM/ESA
Extended Services	VSE/ESA
FFST	VTAM
First Failure Support Technology	WebExplorer
IBM	WebSphere
IMS	WIN-OS/2
IMS/ESA	z/OS
	zSeries

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 UDB documentation library:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Index

Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) 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 http://www.ibm.com/planetwide

Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at http://www.ibm.com/software/data/db2/udb

This site contains the latest information on the technical library, ordering books, product 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

Printed in USA

SC09-4827-01

Spine information:

IBM® DB2 Universal Database™ Programming Server Applications Version 8.2