IBM Db2 V11.5

Routines

2020-08-21

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Contents

Figures

Tables

Chapter 1. Developing Routines

Development of routines is often done when there is no built-in routine available that provides the functionality that is required.

Before you begin

- Read and understand basic routine concepts:
  - See the following topic to learn about types of routines, useful applications of routines, tools for developing routines, routine best practices:
    - "Overview of routines" on page 2
- Learn about the available routine development tools that make it faster and easier to develop routines:
  - See the following topic to learn about the available tools for routine development:
    - "Tools for developing routines" on page 32

About this task

There are different functional types of routines and routine implementations, however the basic steps for developing routines are common for all routines. You must determine what type of routine to create, what implementation to use, define the interface for the routine, develop the routine logic, execute SQL to create the routine, test your routine, and then deploy it for general use.

The following procedure provides steps for getting started with the routine development.

Procedure

1. Determine whether an existing built-in routine already meets your routine needs.
   - If a built-in routine meets your needs, you might want to refer to "Invoking routines" on page 205.
2. Determine what functional type of routine to develop.
3. Determine what routine implementation to use.
   - If a SQL routine is required, refer to the information about SQL routines (SQL PL).
   - If an external routine is required, refer to the information about "External routines" on page 54.

What to do next

The development of SQL and external routines is similar, but there are differences. For both types of routines, you must first design your logic, and then to create the routine in the database but the CREATE statement is specific to the routine type. 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, including the routine name, the number and type of routine parameters, and details about the routine logic. The database manager uses the information that is provided by the clauses to identify and run the routine when it is called. 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 the database system catalog 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.

Because external routines have their logic implemented in user-created libraries or classes that are located in the database file system, additional steps are required to program the logic, build it, and properly locate the resulting library or class file.

Once you have developed routines, you can perform the following steps:

- Debug routines
- Deploy routines to production environments
- Grant execute privileges to users
- Call routines
- Tune the performance of routines

Routines

Routines are database objects that can encapsulate programming and database logic that can be invoked like a programming sub-routine from a variety of SQL interfaces.

Routines can be built in, which means that they are provided with the product, or user-defined, which means that users can create them. Routines can be implemented using SQL statements, a programming language, or a mix of both. Different types of routines provide different interfaces that can be used to extend the functionality of SQL statements, client applications, and some database objects.

For a complete view of the types of routines and implementations that are supported by the database manager, refer to the topic: "Types of routines" on page 3.

Overview of routines

Routines are a type of database object that you can use to encapsulate logic that can be invoked like a programming subroutine. There are many useful applications of routines within a database or database application architecture. You can use routines to improve overall database design, database performance, and data security, as well as to implement basic auditing mechanisms, and more.

Benefits of using routines

Routines are a powerful way of encapsulating logic, which improves the application logic structure, the code maintenance, and potentially improve your application performance.

The following benefits can be gained by using routines:

Encapsulates application logic that can be called from the SQL interface
In an environment that contains many different client applications with common requirements, an effective use of routines can simplify code reuse, standardize the code, and simplify the code maintenance. If a particular aspect of common application behavior is implemented as a routine and changes to the common application behavior is required, only the affected routine that encapsulates the behavior requires modification. If routine is not used to encapsulate the common application logic, all affected application must be modified to address the required change to a common application behavior.

Enable controlled access to other database objects
Routines can be used to control access to database objects. A user might not have permission to generally issue a particular SQL statement, such as the **CREATE TABLE** statement. However, a user can be given required permission to call routines that contain one or more specific implementations of the **CREATE TABLE** statement, thus simplifying privilege management through encapsulation.

Improve application performance by reducing network traffic
When applications are installed on a remote client, each SQL statement is sent separately from the client computer to the database server computer and each result set is returned separately. Sending the query and receiving result set can result in high levels of network traffic. If an application task that requires extensive database interaction can be isolated, you can implement such task as a routine on the server to minimize the network traffic and improve the application performance.

Allow for faster, more efficient SQL execution
Because routines are database objects, they are more efficient at transmitting SQL requests and data than client applications. Therefore, SQL statements within routines can provide better performance than if SQL statements are issued from the client applications. If routines are created with the NOT

FENCED clause, the NOT FENCED routines are run under the database manager process and use the shared memory for communication, which can result in improved application performance.

Allow the interoperability of logic that is implemented in different programming languages
Routines provide high degree of interoperability as it can be called from various applications that are implemented in different programming languages. Client applications in one programming language can call routines that are implemented in a different programming language. For example, C client applications can call .NET common language runtime routines.

Routines can call other routines regardless of the routine type or routine implementation. For example, a Java™ procedure can call an embedded SQL scalar function.

Routines that are created in a database server on one operating system can be called from a database client on a different operating system.

Use of the routines can benefit database administrators, database architects, and database application developers.

The choice of routine type and implementation can affect the degree of the benefits that can be achieved.

Types of routines

There are many different types of routines. Routines can be grouped in different ways, but are primarily grouped by their system or user definitions, by their functionality, and by their implementation.

The supported routine definitions are:

- “Built-in routines” on page 5
- “User-defined routines” on page 5

The supported functional types of routines are:

- “Routines: Procedures” on page 8 (also called stored procedures)
- “Routines: Functions” on page 9
- “Routines: Methods” on page 13

The supported routine implementations are:

- “Built-in routine implementation” on page 18
- “Sourced routine implementation” on page 18
- “SQL routine implementation” on page 18
- “External routine implementation” on page 18

The following diagram illustrates the classification hierarchy of routines. All routines can be either built-in or user-defined. The functional types of routines are in dark grey/blue boxes and the supported routine implementations are in light grey/orange boxes. Built-in routine implementations are emphasized, because this type of implementation is unique.

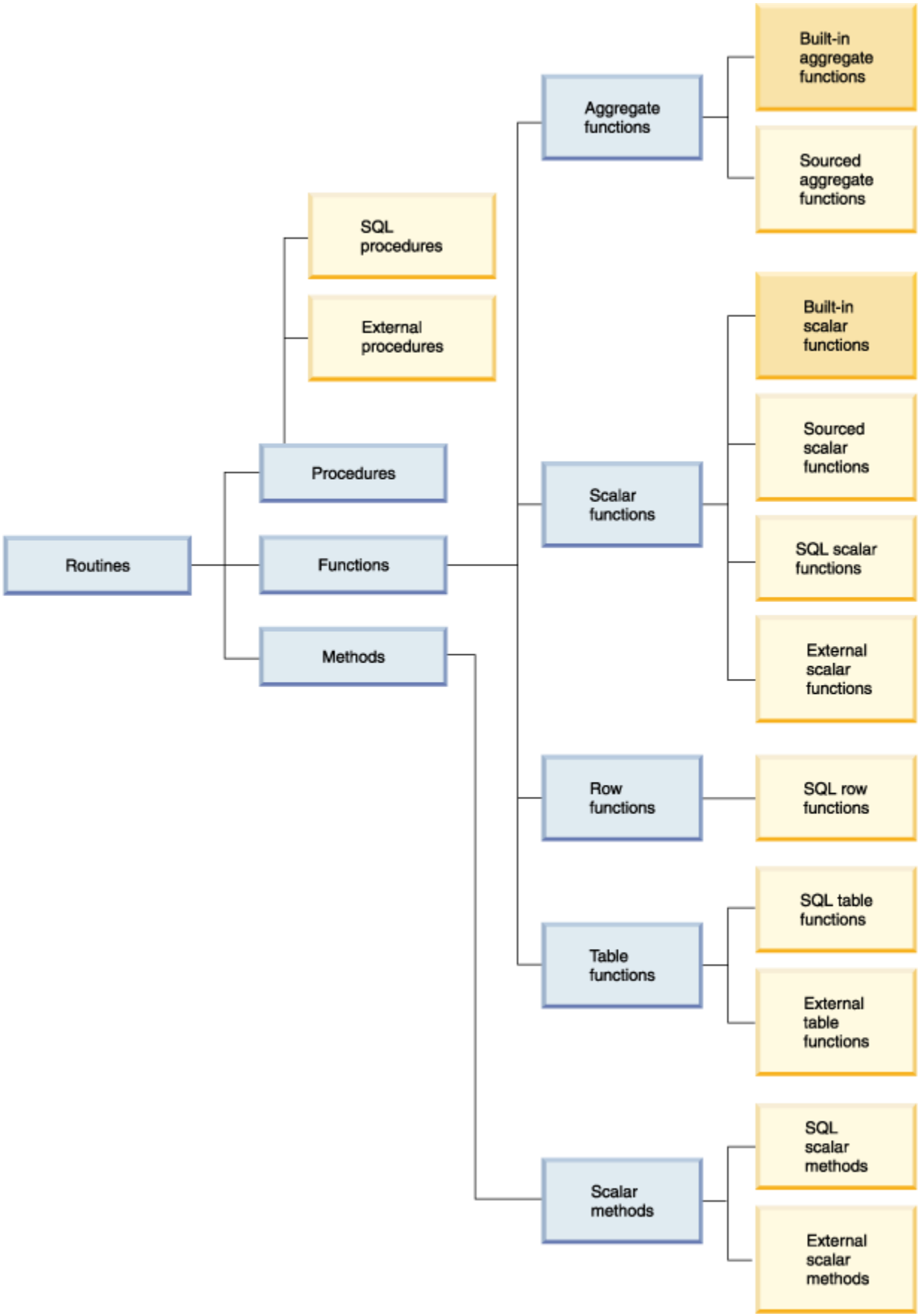


Figure 1. Classifications of routines

The various types of routines provide extensive support for extending SQL language functionality and for developing more modular database applications.

Built-in and user-defined routines
One of the most straight-forward ways of categorizing routines is to divide them into built-in routines and user-defined routines.

Built-in routines are routines that are provided with the product. These routines provide a wide variety of support for tasks ranging from administrative functions to database system and catalog reporting. They are immediately ready-to-use and require no prerequisite setup or registration steps, although users require the necessary privileges to invoke these routines.

User-defined routines are routines that users create themselves. User-defined routines provide a means for users to extend the SQL language beyond the support which is currently available. User-defined routines can be implemented in a variety of ways which include sourcing (re-using the logic of) built-in routines, using SQL statements only, or using SQL with another programming language.

Built-in routines
Built-in routines are routines that are provided with the product. These routines provide a wide variety of routine support for tasks ranging from administrative functions to database system and catalog reporting.

They are characterized by the fact that they are immediately ready-to-use, require no prerequisite setup or routine registration steps, although users require privileges to invoke these routines. These can include built-in routines and are also called SQL Administrative Routines.

Built-in routines provide standard operator support and basic scalar function and aggregate function support. Built-in routines are the first choice of routine that you should use because they are strongly typed and will provide the best performance. Do not create external routines that duplicate the behavior of built-in routines. External routines cannot perform as well as, or be as secure as, built-in routines.

Other built-in routines that you can use are provided with the database in the SYSPROC, SYSFUN, and SYSTOOLS schemas. These routines are essentially SQL and external routines that are defined by the system and provided with the product. Although these additional routines are shipped with the database, they are not built-in routines. Instead they are implemented as pre-installed user-defined routines. These routines typically encapsulate a utility function such as the REBIND_ROUTINE_PACKAGE procedure. You can immediately use these functions and procedures, provided that you have the SYSPROC schema and SYSFUN schema in your CURRENT PATH special register. You can peruse the set of built-in routines if you are considering implementing an external routine that performs administrative functions.

Of particular interest, you might find the ADMIN_CMD procedure useful as it provides a standard interface for executing many popular database commands through an SQL interface.

Built-in routines make it faster and easier for you to implement complex SQL queries and powerful database applications because they are ready-to-use.

User-defined routines
You can create routines to encapsulate logic of your own or use the database provide routines that capture the functionality of most commonly used arithmetic, string, and casting functions. The user-defined routines refer to any procedures, functions, and methods that are created by the user.

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. User-defined functions are also commonly called UDFs.

User-defined routine creation

User-defined procedures, functions and methods are created in the database by executing the appropriate CREATE statement for the routine type. These routine creation statements include:

- CREATE PROCEDURE
- CREATE FUNCTION
- 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. The database manager use 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 the database catalog views that users can query. Executing the CREATE statement to create a routine is also referred to as defining a routine or registering a routine.

User-defined routine definitions are stored in the SYSTOOLS system catalog table schema.

User-defined routine logic implementation

There are three implementation styles that can be used to specify the logic of a routine:

- Sourced: user-defined routines can be *sourced* from the logic of existing built-in routines.
- SQL: user-defined routines can be implemented using only SQL statements.
- External: user-defined routines can be implemented using one of a set of supported programming languages.

  When routines are created in a non-SQL programming language, the library or class built from the code is associated with the routine definition by the value specified in the EXTERNAL NAME clause. When the routine is invoked the library or class associated with the routine is run.

User-defined routines can include a variety of SQL statements, but not all SQL statements.

User-defined routines are strongly typed, but type handling and error-handling mechanisms must be developed or enhanced by routine developers.

After a database upgrade, it may be necessary to verify or update routine implementations.

In general, user-defined routines perform well, but not as well as built-in routines.

User-defined routines can invoke built-in routines and other user-defined routines implemented in any of the supported formats. This flexibility allows users to essentially have the freedom to build a complete library of routine modules that can be re-used.

In general, user-defined routines provide a means for extending the SQL language and for modularizing logic that will be re-used by multiple queries or database applications where built-in routines do not exist.

Comparison of built-in and user-defined routines

Understanding the differences between built-in and user-defined routines can help you determine whether you actually need to build your own routines or whether you can re-use existing routines. The ability to determine when to re-use existing routines and when to develop your own routines can save you time and effort as well as ensure that you are maximizing routine performance.

Built-in routines and user-defined routines differ in a variety of ways. These differences are summarized in the following table:

Table 1. Comparison of built-in and user-defined routines		
Characteristic	**Built-in routines**	**User-defined routines**
Feature support	Extensive numerical operator, string manipulation, and administrative functionality available for immediate use. To use these routines, simply invoke the routines from supported interfaces.	Although not all SQL statements are supported within user-defined routines, a great many are supported. You can also wrap calls to built-in routines within user-defined routines if you want to extend the functionality of the built-in routines. User-defined routines provide a limitless opportunity for routine logic implementation. To use these routines, you must first develop them and then you can invoke them from supported interfaces.
Maintenance	No maintenance is required.	External routines require that you manage the associated external routine libraries.
Upgrade	No or little upgrade impact.	Release to release upgrades might require you to verify your routines.
Performance	Perform better than equivalent user-defined routines.	Generally do not perform as well as equivalent built-in routines.
Stability	Error handling.	Error handling must be programmed by the routine developer.

Whenever it is possible to do so, you should choose to use the built-in routines. These are provided to facilitate SQL statement formulation and application development and are optimized to perform well. User-defined routines give you the flexibility to build your own routines where no built-in routine performs the specific business logic that you want to implement.

Determining when to use built-in or user-defined routines

Built-in routines provide you with time-saving ready-to-use encapsulated functionality whereas *user-defined routines* provide you with the flexibility to define your own routines when no built-in routine adequately contains the functionality that you require.

Procedure

To determine whether to use a built-in or user-defined routine:

1. Determine what functionality you want the routine to encapsulate.
2. Check the list of available built-in routines to see if there are any that meet some or all of your requirements.
   - If there is a built-in routine that meets some, but not all of your requirements:
     – Determine if the functionality that is missing, is functionality that you can add simply to your application? If so, use the built-in routine and modify your application to cover the missing functionality. If the missing functionality is not easily added to your application or if the missing functionality would have to be repeated in many places consider creating a user-defined routine that contains the missing functionality and that invokes the built-in routine.

  - If you expect that your routine requirements will evolve and that you might have to frequently modify the routine definition, consider using a user-defined routine rather than the built-in routine.
  - Determine if there are additional parameters that you might want to pass into or out of the routine. If there are, consider creating a user-defined routine that encapsulates an invocation to the built-in routine.
- If no built-in routine adequately captures the functionality that you want to encapsulate, create a user-defined routine.

Results

To save time and effort, whenever possible consider using built-in routines. There will be times when the functionality that you require will not be available within a built-in routine. For these cases you must create a user-defined routine. Other times it might be possible to include a call to built-in routine from a user-defined routine that covers the extra functionality that you require.

Functional types of routines

There are different functional types of routines. Each functional type provides support for invoking routines from different interfaces for different purposes. Each functional type of routine provides a different set of features and SQL support.

- 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. Procedures can have input, output, and input-output parameters, can execute a wide variety of SQL statements, and return multiple result sets to the caller.
- Functions are relationships between sets 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, expression, or a FROM clause. There are four types of functions: aggregate functions, scalar functions, row functions, and table functions.
- Methods allow you to access user-defined type attributes as well as to define additional behaviors for user-defined 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 as an operation on the attributes of the structured type. For a geometric shape a method might calculate the volume of the shape.

For specific details on each of the functional routine types refer to the topics for each routine type.

Routines: Procedures

Procedures, also called stored procedures, are database objects created by executing the CREATE PROCEDURE statement. Procedures can encapsulate logic and SQL statement and can 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. Procedures can take input, output, and input-output parameters, execute a wide variety of SQL statements, and return multiple result sets to the caller.

Features

- Enable the encapsulation of logic elements and SQL statements that formulate a particular subroutine module
- Can be called from client applications, other routines, triggers, and dynamic compound statements - from anywhere that the CALL statement can be executed.
- Return multiple result-sets
- Support the execution of a large set of SQL statements including SQL statements that read or modify table data in both single and multiple partition databases
- Parameter support for input, output, and input-output parameters
- Nested procedure calls and function invocations are supported

- Recursive calls to procedures are supported
- Savepoints and transaction control are supported within procedures

Limitations

- Procedures cannot be invoked from within SQL statements other than the CALL statement. As an alternative, functions can be used to express logic that transforms column values.
- Output parameter values and result sets of procedure calls cannot be directly used by another SQL statement. Application logic must be used to assign these to variables that can be used in subsequent SQL statements.
- Procedures cannot preserve state between invocations.

Common uses

- Standardization of application logic
  - If multiple applications must similarly access or modify the database, a procedure can provide a single interface for the logic. The procedure is then available for re-use. Should the interface need to change to accommodate a change in business logic, only the single procedure must be modified.
- Isolation of database operations from non-database logic within applications
  - Procedures facilitate the implementation of sub-routines that encapsulate the logic and database accesses associated with a particular task that can be reused in multiple instances. For example, an employee management application can encapsulate the database operations specific to the task of hiring an employee. Such a procedure might insert employee information into multiple tables, calculate the employee's weekly pay based on an input parameter, and return the weekly pay value as an output parameter. Another procedure could do statistical analysis of data in a table and return result sets that contain the results of the analysis.
- Simplification of the management of privileges for a group of SQL statements
  - By allowing a grouping of multiple SQL statements to be encapsulated into one named database object, procedures allow database administrators to manage fewer privileges. Instead of having to grant the privileges required to execute each of the SQL statements in the routine, they must only manage the privilege to invoke the routine.

Supported implementations

- There are built-in procedures that are ready-to-use, or users can create user-defined procedures. The following user-defined implementations are supported for procedures:
  - SQL implementation
  - External implementation

Routines: Functions

Functions are relationships between sets of input data values and a set of result values. They 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
- 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 by using the SUM aggregate function:

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

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 the LENGTH function, and the SUBSTR function. Scalar functions can also be created that do complex mathematical calculations on function 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. The following example shows a query that references the built-in LENGTH scalar function:

```
SELECT lastname, LENGTH(lastname)
FROM employee
```

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.

Table functions

Table functions are functions that for a group of sets of one or more parameters, return 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 operations such as UNION, and any operation that could be applied to a read-only view. The following example demonstrates an SQL table function that updates an inventory table and returns the result set of a query on the updated inventory table:

```
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
```

Functions provide support for the following features:

- Functions are supported across the Db2® brand database products including, among others, Db2, Db2 for z/OS® , and Db2 for IBM® i
- Moderate support for SQL statement execution
- Parameter support for input parameters and scalar or aggregate function return values
- Efficient compilation of function logic into queries that reference functions
- External functions provide support for storing intermediate values between the individual function sub-invocations for each row or value

There are built-in functions that are ready-to-use, or users can create user-defined functions. Functions can be implemented as SQL functions or as external functions. SQL functions can be either compiled or inlined. Inlined functions perform faster than compiled functions, but can execute only a subset of the SQL PL language. See the CREATE FUNCTION statement for more information.

Routines: 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 the LENGTH function, and the SUBSTR function.

Scalar functions can also be created that do complex mathematical calculations on function 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.

Features

- Built-in scalar functions perform well.
- Built-in scalar functions are strongly typed.
- Can be referenced with SQL statements wherever expressions are supported.
- Logic is executed on the server as part of the SQL statement that references it.
- Output of a scalar UDF can be used directly by the statement that references the function.
- When used in predicates, scalar UDF usage can improve overall query performance. When a scalar functions are applied to a set of candidate rows at the server, it can act as a filter, thus limiting the number of rows that must be returned to the client.
- For external scalar user-defined functions, state can be maintained between the iterative invocations of the function by using a scratchpad.

Limitations

- By design, they only return a single scalar value.
- Transaction management is not supported within scalar functions. Commits and rollbacks cannot be executed within scalar function bodies.
- Result sets cannot be returned from scalar functions.
- In a single partition database user-defined external scalar UDFs can contain SQL statements. These statements can read data from tables, but cannot modify data in tables.
- In a multi-partition database environment, user-defined scalar UDFs cannot contain SQL statements.

Common uses

- To manipulate strings within SQL statements.
- To perform basic mathematical operations within SQL statements.
- User-defined scalar functions can be created to extend the existing set of built-in scalar functions. For example, you can create a complex mathematical function, by re-using the existing built-in scalar functions along with other logic.

Supported implementations

- Sourced implementation
- External implementation

Routines: Row functions
A row function is a function, which can only be used with user-defined structured types, 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. Row functions cannot be used in a standalone manner or within SQL statements outside of the context of abstract data types.

Features

- Allows you to map structured type attributes to a row of built-in data type values.

Limitations

- Cannot be used in a standalone manner or in SQL statements outside of the context of user-defined structured types.

Common uses

To make structured type attributes accessible in queries or operations. For example, consider a user-defined structured data type named, 'manager' that extends another structured type person and that has a combination of person attributes and manager specific attributes. If you wanted to refer to these values in a query, you would create a row function to translate the attribute values into a row of values that can be referenced.

Supported implementations

- SQL implementation

Routines: 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.

Features

- Returns a set of data values for processing.
- Can be referenced as part of a SQL query.
- Can make operating system calls, read data from files or even access data across a network in a single partitioned database.
- Results of table function invocations can be directly accessed by the SQL statement that references the table function.
- SQL table functions can encapsulate SQL statements that modify SQL table data. External table functions cannot encapsulate SQL statements.
- For a single table function reference, a table function can be iteratively invoked multiple times and maintain state between these invocations by using a scratchpad.

Limitations

- Transaction management is not supported within user-defined table functions. Commits and rollbacks cannot be executed within table UDFs.
- Result sets cannot be returned from table functions.
- Not designed for single invocations.
- Can only be referenced in the FROM clause of a query.
- User-defined external table functions can read SQL data, but cannot modify SQL data. As an alternative SQL table functions can be used to contain SQL statements that modify SQL data.

Common uses

- Encapsulate a complex, but commonly used sub-query.
- Provide a tabular interface to non-relational data. For example a user-defined external table function can read a spreadsheet and produce a table of values that can be directly inserted into a table or directly and immediately accessed within a query.

Supported implementations

- SQL implementation
- External implementation

Routines: Methods

Methods allow you to access structured type attributes as well as to define additional behaviors 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.

Methods are generally implemented for a structured type to represent operations on the attributes of the structured type. For a geometric shape a method might calculate the volume of the shape. Methods share all of the features of scalar functions.

Features

- Ability to access structured type attributes
- Ability to set structured type attributes
- Ability to create operations on structured type attributes and return a function value
- 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 for typed tables

Common uses

- Create operations on structured types
- Encapsulate the structured type

Supported implementations

There are no built-in methods. Users can create user-defined methods for existing user-defined structured types. Methods can be implemented using one of the following implementations:

- "SQL routine implementation" on page 18
- "External routine implementation" on page 18: C, C++, Java, C# (using OLE API), Visual Basic (using OLE API)

SQL methods are easy to implement, but are generally designed in conjunction with the design of a structured type. External methods provide greater support for flexible logic implementation and allow a user to develop method logic in their preferred programming language.

Comparison of functional types of routines

Understanding the differences between procedures, functions, and methods can help you determine which functional type to implement when building your own routines and can help you determine where and how you can reference existing routines. This can save you time and effort as well as ensure that you are maximizing the functionality and performance of routines.

Procedures, functions, and methods differ in a variety of ways. These differences are outlined in the following table:

Table 2. Comparison of the functional types of routine

Characteristic	Procedures	Functions	Methods
Unique functional characteristics and useful applications	• Enable the encapsulation of logic and SQL statements. • 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. • Nested procedure calls are supported • Recursive procedure calls are supported • Parameter support for input, output, and input-output parameters • Extensive support for SQL statement execution • Can return one or more result-sets • Savepoints and transaction control	• Enable the encapsulation of logic and SQL statements. • Functions are relationships between sets 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. • Moderate support for SQL statement execution. • Parameter support for input parameters and scalar or aggregate function return values. • External functions provide support for storing intermediate values between the individual function sub-invocations for each row or value using a scratchpad. • Efficient compilation of function logic into queries that reference functions.	• Enable the encapsulation of logic and SQL statements. • Methods allow you to access structured type attributes as well as to define additional behaviors for structured types. • Ability to access structured type attributes. • Ability to set structured type attributes. • Ability to create operations on structured type attributes and return a function value.
Functional sub-types of routine	• Not applicable	• Scalar functions • Aggregate functions • Row functions • Table functions	• Not applicable

Table 2. Comparison of the functional types of routine (continued)

Characteristic	Procedures	Functions	Methods
Invocation interface	• Invocation is done through execution of the CALL statement with a reference to the procedure. • Procedure invocation supported wherever CALL statement is supported.	• Invocation is done within an SQL statement within a column select-list, an expression, or in a FROM clause of a select statement, among other locations.	• Invocation is done within an SQL statement that references the structured type associated with the method.
Are there any built-in routines of this type?	• Yes, many. • See the SQL reference for a list of built-in procedures.	• Yes, many. • See the SQL reference for a list of built-in functions.	• No
Supported user-defined routine implementations	• SQL • External – C/C++ (with embedded SQL or CLI API calls) – COBOL – Java (JDBC) – Java (SQLJ) – .NET CLR – OLE: Visual Basic, Visual C++	• SQL • External – C/C++ – Java (JDBC) – Java (SQLJ) – .NET CLR – OLE DB: Visual Basic, Visual C++ (table functions only)	• SQL • External – C – C++
Nested call support	• Yes	• No, however functions are repeatedly invoked for every value in the input set and intermediate values can be stored using a scratchpad.	• No
Performance	• Perform well if routine logic is efficient and best practices are adopted.	• Perform well if routine logic is efficient and best practices are adopted. • Can perform better than a logically equivalent procedure, if the logic only queries data and does not modify data.	• Good performance

Table 2. Comparison of the functional types of routine (continued)			
Characteristic	**Procedures**	**Functions**	**Methods**
Portability	• Highly portable • Particularly portable if SQL implementation is used. • 32-bit and 64-bit external routines supported in a variety of programming languages	• Highly portable • Particularly portable if SQL implementation is used. • 32-bit and 64-bit external routines supported in a variety of programming languages	• Highly portable
Interoperability	• Procedures can call other procedures and can contain SQL statements that invoke functions with SQL access levels less than or equal to the SQL access level of the procedure.	• Functions can contain SQL statements that invoke other functions and can call procedures with SQL access levels less than or equal to the SQL access level of the function.	• Methods can invoke functions with an SQL access level less than or equal to the SQL access level of the method. • Methods cannot call procedures or other methods
Restrictions		• Table functions can only return a single table-reference that must be referenced in the FROM clause of a SELECT statement. output.	

In general the functional characteristics and applications of routines determine what routine type should be used. However, performance and the supported routine implementations also play an important role in determining what routine type should be used.

Determining what functional type of routine to use

Procedures, functions, and methods provide different functional routine and feature support. Determining what routine type to use or implement will determine where and how you can reference and invoke the routine functionality, influence what routine implementations you can use, and can influence what types of functionality your routine can contain.

Before you begin

Determining what routine type is best suited to your needs before beginning to implement it will save you time and possible frustration later.

Read about the functional types of routines to learn about their characteristics.

Procedure

To determine whether to use a procedure, function, or method, do the following:

1. Determine what functionality you want the routine to encapsulate, what interface you want to invoke the routine from, and what routine implementation you want to use.
   - See the following topic:
     – “Comparison of functional types of routines” on page 13

     to determine what functional routine types support these requirements.

2. Determine what SQL statements you want to include in the routine.
   - See the following topic:
     - "SQL statements that can be executed in routines and triggers" on page 33
   - Determine what functional routines support the execution of the required SQL statements.
3. If the routine will only include one or more queries, consider using SQL functions. SQL functions perform well in this situation because they are compiled in-line with the SQL statements that reference them, unlike procedures, which are compiled and invoked separately.
4. Determine whether in the future you might need to extend the functionality of the routine to include functionality of another routine type (for example, procedures support more SQL statements and in general more SQL features than do functions). To avoid having to rewrite a function into a procedure later, consider implementing a procedure now.

Results

In general functional and SQL requirements motivate the choice of what functional type of routine to implement. However, there are cases where it is possible to create logically equivalent routines with different functional types. For example, it is possible to rewrite most basic procedures that return a single result-set as a table function. You can also easily rewrite basic procedures with only a single output parameter as scalar functions.

What to do next

Once you have determined what functional type of routine to use, you might be interested in learning more about routine implementations or in determining what routine implementation to use.

Implementations of routines
Routines can be implemented in a variety of ways. A routine implementation is essentially the underlying form of the routine that contains the logic that is run when a routine is invoked. Understanding the different supported routine implementations can help you understand how routines work and help you determine which routine implementation to choose when implementing user-defined routines.

The available routine implementations include:

- "Built-in routine implementation" on page 18
- "Sourced routine implementation" on page 18
- "SQL routine implementation" on page 18
- "External routine implementation" on page 18

"Built-in routines" on page 5 can be implemented as built-in routines, SQL routines, or external routines. However, their implementation is essentially invisible to the user and in general is of little concern to the user.

"User-defined routines" on page 5 can be implemented as sourced routines, SQL routines, or external routines.

The characteristics of each of the implementations differ and can result in more or less functionality support. Before deciding on a particular implementation, it is a good idea to review the supported functionality and restrictions associated with each implementation, by reading about each of the implementations and then by reading the topic:

- "Comparison of routine implementations" on page 25

A good understanding of the routine implementations can help you make good implementation decisions as well as help you to debug and troubleshoot existing routines.

Built-in routine implementation
Built-in routines are built into the code of the database manager. 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*.

Sourced routine implementation
A routine that is implemented with a sourced routine implementation is one 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. Sourced functions are essentially a special form of an SQL implementation for a function.

SQL routine implementation
A SQL routine implementation is composed entirely of SQL statements.

SQL routine implementations are characterized by the fact that the SQL statements that define the logic of the routines are included within the CREATE statement used to create the routine in the database. SQL routines are quick and easy to implement because of their simple syntax, and perform well due to their close relationship with the database manager.

The 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 variables and condition handlers (DECLARE statement) assigning values to variables (assignment-statement), and for implementing procedural logic (control-statements) such as IF, WHILE, FOR, GOTO, LOOP, SIGNAL, and others. SQL and SQL PL can be used to create SQL procedures, functions, triggers, and compound SQL statements. SQL procedures and functions, along with SQL global variables, user-defined conditions, and data-types, can be grouped together in modules.

External routine implementation
An external routine implementation is one in which the routine logic is defined by programming language code that resides external to the database. As with other routine implementations, routines with external implementations are created in the database by executing a CREATE statement.

The routine logic stored in a compiled library resides on the database server in a special directory path. The association of the routine name with the external code application is asserted by the specification of the EXTERNAL clause in the CREATE statement.

External routines can be written in any of the supported external routine programming languages. Refer to “Supported APIs and programming languages for external routine development” on page 19.

External routine implementation can be somewhat more complex than SQL routine implementation. However, they are extremely powerful because they allow you to 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 the network or file system. For routines that require a smaller degree of interaction with the database, but that must contain a lot of logic or very complex logic, an external routine implementation is a good choice.

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 operate on a VARCHAR data type or a complicated mathematical function that operates on a DOUBLE data type. External routine implementations are also ideal for logic that might involve an external action, such as sending an e-mail.

If you are already comfortable programming in one of the supported external routine programming languages, and need to encapsulate logic with a greater emphasis on programming logic than data access, once you learn the steps involved in creating routines with external implementation, you will soon discover just how powerful they can be.

Supported APIs and programming languages for external routine development
You can develop external routines, including procedures and functions, with specific APIs and associated programming languages.

The following APIs are associated and programming languages are supported.

- ADO.NET
  - .NET Common Language Runtime programming languages
- CLI
- Embedded SQL
  - C
  - C++
  - COBOL (Only supported for procedures)
- JDBC
  - Java
- OLE
  - Visual Basic
  - Visual C++
  - Any other programming language that supports this API.
- OLE DB (Only supported for table functions)
  - Any programming language that supports this API.
- SQLJ
  - Java

Comparison of supported APIs and programming languages for external routine development
It is important to consider the characteristics and limitations of the various supported external routine application programming interfaces (APIs) and programming languages before you start implementing

external routines. This will ensure that you choose the right implementation from the start and that the routine features that you require are available.

Table 3. Comparison of external routine APIs and programming languages

API and programming language	Feature support	Performance	Security	Scalability	Limitations
SQL (includes SQL PL)	• SQL is a high level language that is easy to learn and use, which makes implementatio n go quickly. • SQL Procedural Language (SQL PL) elements allow for control-flow logic around SQL operations and queries.	• Very good. • SQL routines perform better than Java routines. • SQL routines perform as well as C and C ++ external routines created with the NOT FENCED clause.	• Very safe. • SQL procedures always run in the same memory as the database manager. This corresponds to the routine being created by default with the keywords NOT FENCED.	• Highly scalable.	• Cannot access the database server file system. • Cannot invoke applications that reside outside of the database.

Table 3. Comparison of external routine APIs and programming languages (continued)

API and programming language	Feature support	Performance	Security	Scalability	Limitations
Embedded SQL (includes C and C++)	• Low level, but powerful programming language.	• Very good. • C and C++ routines perform better than Java routines. • C and C++ routines created with the NOT FENCED clause perform as well as SQL routines.	• C and C++ routines are prone to programming errors. • Programmers must be proficient in C to avoid making common memory and pointer manipulation errors which make routine implementation more tedious and time consuming. • C and C++ routines should be created with the FENCED clause and the NOT THREADSAFE clause to avoid the disruption of the database manager should an exception occur in the routine at run time. These are default clauses. The use of these clauses can somewhat negatively impact performance, but ensure safe execution. See: Security of routines.	• Scalability is reduced when C and C++ routines are created with the FENCED and NOT THREADSAFE clauses. These routines are run in an isolated **db2fmp** process apart from the database manager process. One **db2fmp** process is required per concurrently executed routine.	• There are multiple supported parameter passing styles which can be confusing. Users should use parameter style SQL as much as possible.

Table 3. Comparison of external routine APIs and programming languages (continued)

API and programming language	Feature support	Performance	Security	Scalability	Limitations
Embedded SQL (COBOL)	• High-level programming language good for developing business, typically file oriented, applications. • Pervasively used in the past for production business applications, although its popularity is decreasing. • COBOL does not contain pointer support and is a linear iterative programming language.	• COBOL routines do not perform as well as routines created with any of the other external routine implementation options.	• No information at this time.	• No information at this time.	• You can create and invoke 32-bit COBOL procedures in 64-bit database instances, however these routines will not perform as well as 64-bit COBOL procedures within a 64-bit database instance.
JDBC (Java) and SQLJ (Java)	• High-level object-oriented programming language suitable for developing standalone applications, applets, and servlets. • Java objects and data types facilitate the establishment of database connections, execution of SQL statements, and manipulation of data.	• Java routines do not perform as well as C and C++ routines or SQL routines.	• Java routines are safer than C and C++ routines, because the control of dangerous operations is handled by the Java Virtual Machine (JVM). This increases reliability and makes it very difficult for the code of one Java routine to harm another routine running in the same process.	• Good scalability • Java routines created with the FENCED THREADSAFE clause (the default) 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.	• To avoid potentially dangerous operations, Java Native Interface (JNI) calls from Java routines are not permitted.

Table 3. Comparison of external routine APIs and programming languages (continued)

API and programming language	Feature support	Performance	Security	Scalability	Limitations
.NET common language runtime supported languages (includes C#, Visual Basic, and others)	• Part of the Microsoft .NET model of managed code. • Source code is compiled into intermediate language (IL) byte code that can be interpreted by the Microsoft .NET Framework common language runtime. • CLR assemblies can be built up from sub-assemblies that were compiled from different .NET programming language source code, which allows users to re-use and integrate code modules written in various languages.	• CLR routines can only be created with the FENCED NOT THREADSAFE clause so as to minimize the possibility of database manager interruption at runtime. This can somewhat negatively impact performance • Use of the default clause values minimizes the possibility of database manager interruption at runtime; however because CLR routines must run as FENCED, they might perform slightly more slowly than other external routines that can be specified as NOT FENCED.	• CLR routines can only be created with the FENCED NOT THREADSAFE clause. They are therefore safe because they will be run outside of the database manager in a separate **db2fmp** process.	• No information available.	• Refer to the topic, "Restrictions on .NET CLR routines".

Table 3. Comparison of external routine APIs and programming languages (continued)

API and programming language	Feature support	Performance	Security	Scalability	Limitations
• 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, and therefore OLE automated routines do not scale well.	• No information available.	• No information available.	• No information available.

Table 3. Comparison of external routine APIs and programming languages (continued)

API and programming language	Feature support	Performance	Security	Scalability	Limitations
• OLE DB	• OLE DB can be used to create user-defined table functions. • OLE DB functions connect to external OLE DB data sources.	• Performance of OLE DB functions depends on the OLE DB provider, however in general OLE DB functions perform better than logically equivalent Java functions, but slower than logically equivalent C, C ++, or SQL 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 the database manager has to process, which can frequently result in improved performance.	• No information available.	• No information available.	• OLE DB can only be used to create user-defined table functions.

Comparison of routine implementations

Understanding the differences between the supported routine implementations can help you determine which routine implementation to use when building your own routines. This can save you time and effort as well as ensure that you are maximizing the functionality and performance of routines.

Built-in, sourced, SQL, and external routine implementations differ in a variety of ways. These differences are outlined in the following table:

Table 4. Comparison of routine implementations

Characteristic	Built-in	Sourced	SQL	External
Features and uses	• Perform very well because their logic is native to the database manager code. • Many common casting, string manipulation, and arithmetic built-in functions are located in the SYSIBM schema.	• Used to provide basic extensions to the functionality of built-in functions. • SQL and SQL PL provide high level programming language support that makes implementing routine logic fast and easy.	• Used to extend the set of built-in functions with more complex functions that can execute SQL statements.	• Developers can program logic in the supported programming language of their choice. • Complicated logic can be implemented. • External actions, actions with impact outside of the database, are directly supported. This can include reading from or writing to the server file system, invoking an application or script on the server, and issuing SQL statements that are not supported in the SQL, sourced, or built-in implementations.
Implementation is built into the database manager code?	• Yes	• No	• No	• No
Supported functional routine types that can have this implementation	• Not applicable	• Functions – Scalar functions – Aggregate functions	• Procedures • Functions • Methods	• Procedures • Functions • Methods
Supported SQL statements	• Not applicable	• Not applicable	• Most SQL statements, including all SQL PL statements, can be executed in routines. • Refer to the topic, "SQL statements that can be executed in routines".	• Many SQL statements, including a sub-set of SQL PL statements, can be executed in routines. • Refer to the topic, "SQL statements that can be executed in routines".

Table 4. Comparison of routine implementations (continued)

Characteristic	Built-in	Sourced	SQL	External
Performance	• Very fast	• In general, about as fast as built-in functions.	• Very good performance if the SQL is efficiently written, database operations are emphasized more than programming logic, and SQL routine best practices are adopted. • Refer to the topic, "SQL routine best practices".	• Very good performance if the programming logic is efficiently written and external routine best practices are adopted. • Refer to the topic, "External routine best practices".
Portability	• Not applicable	• Sourced functions can easily be dropped and recreated in other databases.	• SQL functions can be easily dropped and re-created in other databases.	• External functions can be dropped and re-created in other databases, however care must be taken to ensure that the environment is compatible and that the required supported software is available. • Refer to the topic, "Deploying external routines".
Interoperability	• Not applicable	• They can be referenced wherever built-in functions can be referenced. Sourced functions cannot invoke other functions.	• SQL routines can be referenced in many parts of SQL statements. A SQL routine can invoke other SQL and external routines with SQL access levels that are equal to or less than the SQL access level of the SQL routine.	• External routines can invoke external routines and other SQL routines with SQL access levels that are equal to or less than the SQL access level of the external routine.

In general the functional characteristics and applications of routines determine what routine type should be used. However, performance and the supported routine implementations also play an important role in determining what routine type should be used.

Determining what routine implementation to use

The choice of using or creating a routine with a built-in, sourced, SQL, or external routine implementation can influence what functionality the routine can provide, the performance of the routine, and the likelihood of runtime problems that might require debugging.

About this task

Whenever possible, if there is an existing built-in routine that provides the support that you require, use it. Use existing built-in routines whenever possible. If the functionality you require is very similar to that of an existing built-in function, consider creating a sourced function that extends it.

If you must create a routine, use the following procedure. It is important to determine what routine implementation to use before proceeding too far with routine design.

Procedure

To determine whether to use a sourced, SQL, or external routine implementation when creating a routine:

1. Determine whether you want to create a procedure, function, or method. This should always be your first step when developing a routine. Also determine what are the support implementations for that routine type. See:
   - "Comparison of functional types of routines" on page 13
2. Determine what SQL statements you want to include in the routine. The set of SQL statements that you want to execute in a routine can limit your choice of routine implementation. See:
   - "Determining what SQL statements can be executed in routines" on page 39
3. Determine if now or in the future the routine logic must access data, files, or applications that reside external to the database. The data, files, or applications might reside in the file system of the database server or in the available network.
   - If the routine logic must access entities outside of the database, you must use an external routine implementation.
4. Determine the number of queries to be included in the routine relative to the quantity of procedural flow logic.
   - If the routine logic contains primarily procedural flow logic and very few queries, create an external routine.
   - If the routine logic contains many queries and a minimal amount of procedural flow logic, create an SQL routine.

Usage of routines

Routines can be used to solve many common problems faced by database architects, database administrators, and application developers alike. They can help improve the structure, maintenance, and performance of your applications.

The following list provides some examples of scenarios in which you might use routines:

- Administering databases with routines
- Extending SQL function support with user-defined functions
- Auditing data changes using routines and other SQL features

Administering databases with built-in routines

You can easily administer databases in your applications with use of database built-in routines.

About this task

You can use a set of database built-in procedures and function to perform administrative tasks. For example, database built-in procedures can perform the following tasks:

- Run a database command through an SQL interface

- Modify configuration parameters
- Manage packages
- Run snapshot-related tasks

Database built-in procedures and functions are found in the SYSPROC, SYSFUN, and SYSTOOLS schemas. Your application can implement database built-in routines to perform administrative tasks or if you want to access the results of administrative tasks through an SQL interface. Your application can filter, sort, modify, or reuse the results of administrative tasks in another query without the need to create your own routines.

The ADMIN_CMD command along with other built-in routines provide comprehensive administration support.

ADMIN_CMD for running database commands through a SQL interface

You can use the ADMIN_CMD routine to execute database commands through an SQL interface. The ADMIN_CMD routine accepts a database command with appropriate options and values as a string argument. The ADMIN_CMD routine runs the database command that is contained in the argument string and returns the results in a tabular or scalar format that can be used as part of a larger query or operation.

Built-in administrative routines

Database built-in routines can be used from the CLP or database applications wherever invocation of the specified routine is supported. Examples of built-in routines include the following routines:

- ADMIN_CMD
- MON_GET_CONNECTION
- MON_GET_DATABASE
- MON_GET_TABLE
- REBIND_ROUTINE_PACKAGE

The ADMIN_CMD routine and the other built-in routines are available for use if you have the SYSPROC, SYSFUN, and SYSTOOLS schema names included in the **CURRENT PATH** special register value, which they are by default.

For examples of a built-in routine usage, refer to the built-in routine specific reference documentation.

Extension of SQL function support with user-defined functions

If no built-in functions encapsulate the logic that you require, you can create your own user-defined functions. User-defined functions are a great way of extending the basic set of SQL functions.

About this task

Whether you or a group of users need a function to implement a complex mathematical formula, specific string manipulation, or to do some semantic transformations of values, you can easily create a high-performance SQL function to do this that can be referenced like any existing built-in SQL function.

For example, consider a user that requires a function that converts a value in one monetary currency to another monetary currency. Such a function is not available within the set of built-in routines. This function can be created however as a user-defined SQL scalar function. Once created this function can be referenced wherever scalar functions are supported within an SQL statement.

Another user might require a more complex function that sends an e-mail whenever a change is made to a particular column in a table. Such a function is not available within the set of built-in routines. This function can be created however as a user-defined external procedure with a C programming language implementation. Once created, this procedure can be referenced wherever procedures are supported, including from within triggers.

These examples demonstrate how easily you can extend the SQL language by creating user-defined routines.

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.

Before you begin

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 of rows can be inserted into and stored in a separate table as a history of the changes made to the table.

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

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
```

Tools for developing routines

There are various development environments and tools available for developing procedures and functions.

Some of these tools are provided with the database whereas others are integrated components within popular integrated development environments. There are graphical and non-graphical interfaces and tools that can be used to develop procedures and functions.

The following command line interface can be used for developing routines in database servers:

- Db2 Command Line Processor (CLP)

Several IBM software products provide graphical tools for developing routines in database servers including, but not limited to:

- IBM Data Studio
- IBM Distributed Unified Debugger
- IBM Rational® Application Developer
- IBM Rational Web Developer
- IBM WebSphere® Studio

Some database features can be used to add graphical tool support for developing routines from within a software provided by other vendors, including:

- IBM Database Add-Ins for Microsoft Visual Studio

IBM Data Studio routine development support

IBM Data Studio provides an easy-to-use development environment for creating, building, debugging, testing, and deploying stored procedures.

IBM Data Studio provides graphical tools which simplify the process of creating routines by allowing you to focus on the stored procedure logic rather than the details of generating the basic CREATE statement, building, and installing stored procedures on a database server. Additionally, you can develop stored procedures on one operating system and build them on other server operating systems.

IBM Data Studio is a graphical application that supports rapid development. You can use it to perform the following tasks:

- Create new stored procedures.
- Build stored procedures on local and remote database servers.
- Modify and rebuild existing stored procedures.
- Test and debug the execution of installed stored procedures. (Note: to utilize the debugging facilities offered by IBM Data Studio, the debugging user must be a member of the SYSDEBUG role).

IBM Data Studio allows you to manage your work in projects. Each IBM Data Studio project saves your connections to specific databases. In addition, you can create filters to display subsets of the stored procedures on each database. When opening a new or existing IBM Data Studio project, you can filter stored procedures so that you view them based on their name, schema, language, or collection ID (for Db2 for z/OS servers).

For more information about the IBM Data Studio product, see IBM Knowledge Center: http://www-01.ibm.com/support/knowledgecenter/SS62YD/product_welcome.html

SQL statements that can be executed in routines and triggers

Successful execution of SQL statements in routines is subject to restrictions and conditional on certain prerequisites being met. However, it is possible to execute many SQL statements in routines and triggers.

If a statement invokes a routine, the effective SQL data access level for the statement will be the greater of:

- The SQL data access level of the statement from the following table.
- The SQL data access level of the routine specified when the routine was created.

For example, the CALL statement has an SQL data access level of CONTAINS SQL. However, if a stored procedure defined as READS SQL DATA is called, the effective SQL data access level for the CALL statement is READS SQL DATA.

The following table lists all supported SQL statements, including SQL PL control-statements, and identifies if each SQL statement can be executed within the various types of routines. For each SQL statement listed in the first column, each of the subsequent columns shows an X to indicate if the statement is executable within the routine. The final column identifies the minimum SQL access level required to allow the statement execution to succeed. When a routine invokes an SQL statement, the effective SQL data access indication for the statement must not exceed the SQL data access indication declared for the routine. For example, a function defined as READS SQL DATA could not call a procedure defined as MODIFIES SQL DATA. Unless otherwise noted in a footnote, all of the SQL statements may be executed either statically or dynamically.

Table 5. SQL statements that can be executed in routines

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
ALLOCATE CURSOR	X		X	X	MODIFIES SQL DATA
ALTER {BUFFERPOOL, DATABASE PARTITION GROUP, FUNCTION, METHOD, NICKNAME, PROCEDURE, SEQUENCE, SERVER, TABLE, TABLESPACE, TYPE, USER MAPPING, VIEW}			X	X	MODIFIES SQL DATA
ASSOCIATE LOCATORS	X				
AUDIT			X	X	MODIFIES SQL DATA
BEGIN DECLARE SECTION			X	X	NO SQL(3)
CALL	X	X	X	X	CONTAINS SQL(12)

Table 5. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
CASE	X	X			CONTAINS SQL
CLOSE	X		X	X	READS SQL DATA
COMMENT ON	X		X	X	MODIFIES SQL DATA
COMMIT	X(6)		X(6)		MODIFIES SQL DATA
Compound SQL	X	X	X	X	CONTAINS SQL
CONNECT(2)					
CREATE {ALIAS, BUFFERPOOL, DATABASE PARTITION GROUP, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING, GLOBAL TEMPORARY TABLE(11), INDEX(11), INDEX EXTENSION, METHOD, NICKNAME, PROCEDURE, SCHEMA, SEQUENCE, SERVER, TABLE(11), TABLESPACE, TRANSFORM, TRIGGER, TYPE, TYPE MAPPING, USER MAPPING, VIEW(11), WRAPPER }	X (8)		X		MODIFIES SQL DATA
DECLARE CURSOR	X	X	X		NO SQL(3)
DECLARE GLOBAL TEMPORARY TABLE	X		X	X	MODIFIES SQL DATA

Table 5. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
DELETE	X	X	X	X	MODIFIES SQL DATA
DESCRIBE(9)			X	X	READS SQL DATA
DISCONNECT(4)					
DROP	X(8)		X	X	MODIFIES SQL DATA
END DECLARE SECTION			X	X	NO SQL(3)
EXECUTE	X		X	X	CONTAINS SQL(5)
EXECUTE IMMEDIATE	X		x	X	CONTAINS SQL(5)
EXPLAIN	X		X	X	MODIFIES SQL DATA
FETCH	X		X	X	READS SQL DATA
FLUSH EVENT MONITOR			X	X	MODIFIES SQL DATA
FLUSH PACKAGE CACHE			X	X	MODIFIES SQL DATA
FOR	X	X			READS SQL DATA
FREE LOCATOR			X	X	CONTAINS SQL
GET DIAGNOSTICS	X	X			READS SQL DATA
GOTO	X	X			CONTAINS SQL
GRANT	X		X	X	MODIFIES SQL DATA
IF	X	X			CONTAINS SQL
INCLUDE			X	X	NO SQL
INSERT	X	X	X	X	MODIFIES SQL DATA
ITERATE	X	X			CONTAINS SQL
LEAVE	X	X			CONTAINS SQL
LOCK TABLE	X		X	X	CONTAINS SQL
LOOP	X	X			CONTAINS SQL

Table 5. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
MERGE	X	X	X	X	MODIFIES SQL DATA
OPEN	X		X	X	READS SQL DATA(7)
PREPARE	X		X	X	CONTAINS SQL
REFRESH TABLE			X	X	MODIFIES SQL DATA
RELEASE(4)					
RELEASE SAVEPOINT	X		X	X	MODIFIES SQL DATA
RENAME TABLE			X	X	MODIFIES SQL DATA
RENAME TABLESPACE			X	X	MODIFIES SQL DATA
REPEAT	X	X			CONTAINS SQL
RESIGNAL	X				MODIFIES SQL DATA
RETURN	X				CONTAINS SQL
REVOKE			X	X	MODIFIES SQL DATA
ROLLBACK(6)	X		X		
ROLLBACK TO SAVEPOINT	X		X	X	MODIFIES SQL DATA
SAVEPOINT	X				MODIFIES SQL DATA
select-statement	X		X	X	READS SQL DATA
SELECT INTO	X		X(10)	X(10)	READS SQL DATA(7)
SET CONNECTION(4)					
SET INTEGRITY			X		MODIFIES SQL DATA
SET special register	X	X	X	X	CONTAINS SQL
SET variable	X	X			CONTAINS SQL
SIGNAL	X	X			MODIFIES SQL DATA

Table 5. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
TRANSFER OWNERSHIP			X	X	MODIFIES SQL DATA
TRUNCATE			X	X	MODIFIES SQL DATA
UPDATE	X	X	X		MODIFIES SQL DATA
VALUES INTO	X		X	X	READS SQL DATA
WHENEVER	X		X		NO SQL(3)
WHILE	X	X			

Note:

1. Compound SQL (compiled) statements can be used as the body of SQL procedures, SQL functions, triggers, or as stand-alone statements.
2. Compound SQL (inline) statements can be used as the body of SQL functions, SQL methods, triggers, or as stand-alone statements.
3. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
4. Connection management statements are not allowed in any routine execution context.
5. This situation depends on the statement being executed. The statement specified for the EXECUTE statement must be allowed in the context of the particular SQL access level in effect. For example, if the SQL access level READS SQL DATA is in effect, the statement cannot be INSERT, UPDATE, or DELETE.
6. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) can be used in a stored procedure, but only if the stored procedure is called directly from an application, or indirectly through nested stored procedure calls from an application. If any trigger, function, method, or atomic compound statement is in the call chain to the stored procedure, a COMMIT or a ROLLBACK of a unit of work is not allowed.
7. If the SQL access level READS SQL DATA is in effect, no SQL data change statement can be embedded in the SELECT INTO statement or in the cursor referenced by the OPEN statement.
8. SQL routines can only issue CREATE and DROP statements statically for indexes, tables, and views. For other objects, CREATE and DROP statements can be issued dynamically using the following statements:
   - EXECUTE IMMEDIATE
   - PREPARE, followed by EXECUTE
9. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command.
10. This is only supported for embedded SQL routines.
11. When referenced in an SQL procedure, the statement can only be executed statically.
12. The procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a procedure defined as MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL. A routine defined as CONTAINS SQL can call a procedure defined as CONTAINS SQL or NO SQL. The arguments specified

for the procedure might also require a different data access level. For example, a scalar fullselect as an argument would require the data access level for the statement to be READS SQL DATA.

Errors

Table 5 on page 33 indicates whether the SQL statement specified by the first column is allowed to execute inside a routine that has the specified SQL data access level. If the statement exceeds the data access level, an error is returned when the routine is executed.

- If an executable SQL statement is encountered inside a routine defined with the NO SQL data access level, then SQLSTATE 38001 is returned.
- For other execution contexts, the SQL statements that are unsupported in any context return an SQLSTATE 38003 error.
- For other SQL statements that are not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned.
- In a READS SQL DATA context, SQLSTATE 38002 is returned.
- During creation of an SQL routine, a statement that does not match the SQL data access level returns an SQLSTATE 42985 error.

SQL access levels in routines

The degree to which routines can execute SQL statements is determined by the SQL access level of the routine. The SQL access level for a routine is determined by both what is permitted for the particular type of routine and what limitation is specified explicitly within the CREATE statement that defines a routine.

The SQL access levels follow:

- NO SQL
- CONTAINS SQL
- READS SQL
- MODIFIES SQL

This SQL access level 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.

The default and maximal SQL access levels for different types of routines are shown in the following table:

Table 6. Default and maximum SQL access levels for routines

Routine type	Default SQL access level	Maximum allowed SQL access level
SQL procedures	MODIFIES SQL DATA	MODIFIES SQL DATA
SQL functions (scalar functions)	READS SQL DATA	READS SQL DATA
SQL functions (table functions)	READS SQL DATA	MODIFIES SQL DATA
External procedures	MODIFIES SQL DATA	MODIFIES SQL DATA
External functions (scalar functions)	READS SQL DATA	READS SQL DATA
External functions (table functions)	READS SQL DATA	READS SQL DATA

Optimal performance of routines is achieved when the most restrictive SQL access clause that is valid is specified in the routine CREATE statement.

In the CREATE statement for a routine:

- If you explicitly specify READS SQL DATA, no SQL statement in the routine can modify data.

- If you explicitly specify CONTAINS SQL DATA, no SQL statement in the routine can modify or read data.
- If you explicitly specify NO SQL, there must be no executable SQL statements in the routine.

Determining what SQL statements can be executed in routines
Many, but not all SQL statements can be executed in routines. Execution of a particular SQL statement within a routine is dependent on the type of routine, the implementation of the routine, the maximum SQL access level specified for the routine, and the privileges of the routine definer and invoker.

Before you begin

Determining what SQL statements can be executed within a routine before you implement your routine can ensure that you make the right choice of routine type and implementation from the start.

To successfully execute a SQL statement in a routine, the following prerequisites must be met:

- The SQL access level of the routine must permit the execution of the particular SQL statement.
  - The SQL access level of a routine is specified in the CREATE statement for the routine.
  - Some SQL access levels are not supported for certain types of routines. Refer to the following restrictions.
- The routine definer must have the necessary privileges to execute the SQL statement.
  - The privileges required to execute every supported SQL statement are provided in the SQL Reference.
- No other separate restriction restricts the execution of the statement.
  - Refer to the SQL Reference for a list of restrictions specific to the given SQL statement.

Restrictions

The following restrictions limit the set of SQL statements that can be executed within routines. In particular these restrictions limit what SQL access levels can be specified for particular types of routines:

- External functions cannot be specified with the MODIFIES SQL DATA access level.
- External procedures that will be called from a trigger cannot be specified with a MODIFIES SQL DATA access level.

Procedure

To determine what SQL statements can be invoked in a particular routine:

1. Determine the SQL access level of the routine. If it is an existing routine, examine the CREATE statement that was used to create the routine. The SQL access level clause might be explicitly defined in the DDL as one of: NO SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA. If no such clause is explicitly specified, then the default value for the routine is implied.
   - For SQL procedures the default is MODIFIES SQL DATA.
   - For SQL functions the default is MODIFIES SQL DATA.
   - For external procedures the default is MODIFIES SQL DATA.
   - For external functions the default is READS SQL DATA.
2. Refer to the table in the topic, "SQL statements that can be executed in routines". Look up the SQL statement of interest by name.
3. Check if the SQL statement is supported for the specific type of routine and implementation.
4. Verify that the required SQL access level to execute the statement matches the SQL access level of the routine.
5. Carefully read any usage notes or footnotes to ensure that there are no other restrictions on the SQL statement execution.

Results

If the SQL statement is indicated as being executable within a routine, the routine SQL access level meets the prerequisites for executing the statement within the routine, and all other prerequisites have been met, the SQL statement should be successfully executable from the routine.

Portability of routines

Routine portability refers to the ease with which a routine can be deployed. Portability comprises such factors as operating system compatibility, runtime environment compatibility, software compatibility, invocation interface compatibility as well as other routine implementation factors such as compatibility of support for the SQL statements executed within a routine.

Routine portability is essential if the environment to which you will deploy a routine is not identical to the environment in which the routine was developed. In general routines are highly portable between operating systems and even between the various database products and editions. It is a good idea to consider the potential portability problems before you begin developing routines so that you minimize the likelihood of rework later.

The following topics include information related to factors that can limit the portability of routines:

- Supported database product editions
- Supported development and compiler software
- SQL statements that can be executed in routines
- Restrictions on routines
- Deploying routines

Interoperability of routines

The interoperability of routines of different types and with different programming implementations ensures that routines can be highly re-useable modules throughout the life-span of a database system.

Because code modules are often implemented by different programmers 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, the routine infrastructure is designed to support a high degree of routine interoperability.

Interoperability of routines is characterized by the ability to reference and invoke routines of different types and implementations from other routines seamlessly and without any additional requirements. Routines are interoperable in the following ways:

- 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 Java 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 database 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 benefits listed previously 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.

Performance of routines

The performance of routines is impacted by a variety of factors including the type and implementation of the routine, the number of SQL statements within the routine, the degree of complexity of the SQL in the routine, the number of parameters to the routine, the efficiency of the logic within the routine implementation, the error handling within the routines and more.

Because users often choose to implement routines to improve the performance of applications, it is important to get the most out of routine performance.

The following table outlines some of the general factors that impact routine performance and gives recommendations on how to improve routine performance by altering each factor. For further details on performance factors that impact specific routine types, refer to the performance and tuning topics for the specific routine type.

Table 7. Performance considerations and routine performance recommendations

Performance consideration	Performance recommendation
Routine type: procedure, function, method	• Procedures, functions, and methods serve different purposes and are referenced in different places. Their functional differences make it difficult to compare their performance directly. • In general procedures can sometimes be rewritten as functions (particularly if they return a scalar value and only query data) and enjoy slight performance improvements, however these benefits are generally a result of simplifying the SQL required to implement the SQL logic. • User-defined functions with complex initializations can make use of scratchpads to store any values required in the first invocation so that they can be used in subsequent invocations.
Routine implementation: built-in or user-defined	• For equivalent logic, built-in routines perform the best, followed by built-in routines, because they enjoy a closer relationship with the database engine than do user-defined routines. • User-defined routines can perform very well if they are well coded and follow best practices.

Table 7. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Routine implementation: SQL or external routine implementation	• SQL routines are more efficient than external routines because they are executed directly by the database server. • SQL procedures generally perform better than logically equivalent external procedures. • For simple logic, SQL function performance will be comparable to that of an equivalent external function. • For complex logic, such as math algorithms and string manipulation functions that require little SQL, it is better to use an external routine in a low level programming language such as C because there is less dependence on SQL support. • See Comparison of routine implementations for a comparison of the features, including performance, of the supported external routine programming language options.
External routine implementation programming language	• See: Comparison of external routine APIs and programming languages for a comparison of the performance features that you should consider when selecting an external routine implementation. • Java (JDBC and SQLJ APIs) – Java routines with very large memory requirements are best created with the FENCED NOT THREADSAFE clause specified. Java routines with average memory requirements can be specified with the FENCED THREADSAFE clause. – For fenced threadsafe Java routine invocations, the database manager attempt 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 multi-threaded Java **db2fmp** processes. FENCED THREADSAFE routines, in contrast, perform better because they can share a small number of JVMs. • C and C++ – In general C and C++ routines perform better than other external routine implementations and as well as SQL routines. – For deployment of routines to both 32-bit and 64-bit database instances, compile C and C++ routines in 32-bit format. • COBOL – In general COBOL performance is good, but COBOL is not a recommended routine implementation.

Table 7. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Number of SQL statements within the routine	• Routines should contain more than one SQL statement, otherwise the overhead of routine invocation is not performance cost effective. • Logic that must make several database queries, process intermediate results, and ultimately return a subset of the data that was worked with is the best logic for routine encapsulation. Complex data mining, and large updates requiring lookups of related data are examples of this type of logic. Heavy SQL processing is done on the database server and only the smaller data result set is passed back to the caller.
Complexity of SQL statements within the routine	• It makes good sense to include very complex queries within your routines so that you capitalize on the greater memory and performance capabilities of the database server. • Do not worry about the SQL statements being overly complex.
Static or dynamic SQL execution within routines	• In general static SQL performs better than dynamic SQL. In routines there are no additional differences when you use static or dynamic SQL.
Number of parameters to routines	• Minimizing the number of parameters to routines can improve routine performance as this minimizes the number of buffers to be passed between the routine and routine invoker.
Data types of routine parameters	• You can improve the performance of routines by using VARCHAR parameters instead of CHAR parameters in the routine definition. Using VARCHAR data types instead of CHAR data types prevents the database manager 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, the database manager 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 the database manager simply passing the 14 character string and a null terminator across the network.

Table 7. Performance considerations and routine performance recommendations (continued)	
Performance consideration	**Performance recommendation**
Initialization of parameters to routines	• It is a good idea to always initialize input parameters to routines, particularly if the input routine parameter values are null. For null value routine parameters, a shorter or empty buffer can be passed to the routine instead of a full sized buffer, which can improve performance.
Number of local variables in routines	• Minimizing the number of local variables declared within a routine can improve performance by minimizing the number of SQL statements executed within the routine. • In general aim to use as few variables as possible. Re-use variables if this will not be semantically confusing.
Initialization of local variables in routines	• If possible, it is a good practice to initialize multiple local variables within a single SQL statement as this saves on the total SQL execution time for the routine.
Number of result sets returned by procedures	• If you can reduce the number of result sets returned by a routine you can improve routine performance.
Size of result sets returned by routines	• Make sure that for each result set returned by a routine, the query defining the result filters the columns returned and the number of rows returned as much as possible. Returning unnecessary columns or rows of data is not efficient and can result in sub-optimal routine performance.
Efficiency of logic within routines	• As with any application, the performance of a routine can be limited by a poorly implemented algorithm. Aim to be as efficient as possible when programming routines and apply generally recommended coding best practices as much as possible. • Analyze your SQL and wherever possible reduce your query to its simplest form. This can often be done by using CASE expressions instead of CASE statements or by collapsing multiple SQL statements into a single statement that uses a CASE expression as a switch.

Table 7. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	**NOT FENCED clause usage:** • In general, creating your routine with the NOT FENCED clause, which makes it runs in the same process as the database manager, is preferable over creating it with the FENCED clause, which makes it run in a special database process outside of the engine's address space. • While you can expect improved routine performance when running routines as not fenced, user code in unfenced routines can accidentally or maliciously corrupt the database or damage the database control structures. You should only use the NOT FENCED clause when you need to maximize 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, refer to Security of routines. If the routine is not safe enough to run in the database manager's process, use the FENCED clause when creating the routine. To limit the creation and running of potentially unsafe code, the database manager 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.

Table 7. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	**FENCED THREADSAFE clause usage** • Routines created with the FENCED THREADSAFE clause 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 abnormally ends, only the thread running this routine is terminated. Other routines in the process continue running. However, the failure that caused this thread to abnormally end 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 Linux® and UNIX operating systems can see their Java and C THREADSAFE processes by looking for **db2fmp** (Java) or **db2fmp** (C).

Table 7. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	**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 Linux and UNIX operating systems, 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**.
Level of SQL access in routine: NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA	• Routines that are created with a lower level of SQL access clause will perform better than routines created with a higher level of SQL access clause. Therefore you should declare your routines with the most restrictive level of SQL access clause. For example, if your routine only reads SQL data, do not create it with the MODIFIES SQL DATA clause, but rather create it with the more restrictive READS SQL DATA clause.
Determinism of routine (DETERMINISTIC or NOT DETERMINISTIC clause specification)	• Declaring a routine with the DETERMINISTIC or NOT DETERMINISTIC clause has no impact on routine performance.
Number and complexity of external actions made by routine (EXTERNAL ACTION clause specification)	• Depending on the number of external actions and the complexity of external actions performed by an external routine, routine performance can be hindered. Factors that contribute to this are network traffic, access to files for writing or reading, the time required to execute the external action, and the risk associated with hangs in external action code or behaviors.
Routine invocation when input parameters are null (CALLED ON NULL INPUT clause specification)	• If receiving null input parameter values results in no logic being executed and an immediate return by the routine, you can modify your routine so that it is not fully invoked when null input parameter values are detected. To create a routine that ends invocation early if routine input parameters are received, create the routine and specify the CALLED ON NULL INPUT clause.

Table 7. Performance considerations and routine performance recommendations (continued)	
Performance consideration	**Performance recommendation**
Procedure parameters of type XML	• The passing of parameters of data type XML is significantly less efficient in external procedures implemented in either the C or JAVA programming language than in SQL procedures. When passing one or more parameters of data type XML, consider using SQL procedures instead of external procedures. • XML data is materialized when passed to stored procedures as IN, OUT, or INOUT parameters. If you are using Java stored procedures, the heap size (**java_heap_sz** configuration parameter) might need to be increased based on the quantity and size of XML arguments, and the number of external stored procedures that are being executed concurrently.

Once routines are created and deployed, it might be harder to determine what environmental and routine specific factors are impacting routine performance, and hence it is important to design routines with performance in mind.

Security of routines

The security of routines is paramount to ensure their continued functioning, to minimize the risk of tampering, and to protect the database system environment. There are a few categories of routine security considerations each with varying levels of risk. One must be aware of these risks when developing or maintaining routines so as to mitigate unfortunate outcomes as much as possible.

Security control of who can create routines

The security of routines begins when users are given the necessary privileges to execute the CREATE statement required to create routines in the database. When granting these privileges, it is important to understand the corresponding risks:

- Users with the privilege to execute the CREATE statement for a routine can create multiple routines.
- Users with the privilege to execute the CREATE statement for a routine can create routines that can modify the database layout or database data subject to the other privileges that user has.
- Users that successfully create routines are automatically granted the EXECUTE privilege required to invoke the routine.
- Users that successfully create routines are automatically granted the ALTER ROUTINE privilege required to modify the routine.

To minimize the risk of users modifying the database and data:

- Minimize the number of users that have the privilege to create routines.
- Ensure that the user IDs of departed employees are removed, or if they are re-used, be sure to assess the procedure related privileges.

Refer to the topics on controlling access to database objects and data for more on how to grant and revoke privileges from one, many, or all database users.

Security control of who can invoke routines

It is easy to determine when users require privileges: they are unable to do something. It is harder to determine when users no longer require these privileges. This is particularly true when it comes to users with privileges to invoke routines, as allowing them to retain their privileges can introduce risks:

- Users that have been granted the EXECUTE privilege to invoke a routine will continue to be able to invoke the routine until this privilege is removed. If the routine contains sensitive logic or acts on sensitive data this can be a business risk.

To minimize the risk of users modifying the database and data:

- Minimize the number of users that have the privilege to invoke routines.
- Ensure that the user IDs of departed employees are removed, or if they are re-used, be sure to assess the procedure related privileges.
- If you suspect that someone is maliciously invoking routines, you should revoke the EXECUTE privilege for each of those routines.

Security control of routines defined with FENCED or NOT FENCED clauses

When formulating the CREATE statement for a routine, you must determine whether you want to specify the FENCED clause or NOT FENCED clause. Once you understand the benefits of creating a routine as fenced or unfenced it is important to assess the risks associated with running routines with external implementations as NOT FENCED.

- Routines created with the NOT FENCED clause can accidentally or maliciously corrupt the database manager's shared memory, damage the database control structures, or access database manager resources which can cause the database manager to fail. There is also the risk that they will corrupt databases and their tables.

To ensure the integrity of the database manager and its databases:

- Thoroughly screen routines you intend to create that specify the NOT FENCED clause. These routines must be fully tested, debugged, and not exhibit any unexpected side-effects. In the examination of the routine code, 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 languages.

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.

Securing routines

When creating routines it is important to ensure that the routines, routine libraries (in the case of external routines), and the privileges of the users that will interact with the routines are managed with routine security in mind.

Before you begin

Although it might not be necessary to have anything as elaborate as a routine security strategy, it helps to be mindful of the factors contributing to the security of routines and to follow a disciplined approach when securing routines.

- Read the topic, "Security of routines".
- To fully secure routines within the database system you must have:
  - Root user access on the database server operating system.
  - One of the SECADM or ACCESSCTRL authorities.

About this task

Whether you are creating a routine, or assessing an existing routine, the procedure for securing a routine is similar.

Procedure

1. Limit the number of user IDs with the privileges required to create routines and ensure that these users are allowed to have these privileges.
   - Upon successful execution of the CREATE statement for a routine, this user ID will automatically be granted other privileges including the EXECUTE privilege, which allows the user to invoke the routine, and the GRANT EXECUTE privilege, which allows the user to grant the ability to invoke the routine to other users.
   - Ensure that the users with this privilege are few and that the right users get this privilege.
2. Assess the routine for potentially malicious or inadequately reviewed or tested code.
   - Consider the origin of the routine. Is the party that supplied the routine reliable?
   - Look for malicious code such as code that attempts to read or write to the database server file system and or replace files there.
   - Look for poorly implemented code related to memory management, pointer manipulation, and the use of static variables that might cause the routine to fail.
   - Verify that the code has been adequately tested.
3. Reject routines that appear to be excessively unsafe or poorly coded - the risk is not always worth it.
4. Contain the risks associated with only somewhat potentially risky routines.
   - SQL user-defined SQL routines are by default created as NOT FENCED THREADSAFE routines, because they are safe to run within the database manager memory space. For these routines you do not need to do anything.
   - Specify the FENCED clause in the CREATE statement for the routine. This will ensure that the routine operation does not affect the database manager. This is a default clause.
   - If the routine is multi-threaded, specify the NOT THREADSAFE clause in the CREATE statement for the routine. This will ensure that any failures or malicious code in the routine do not impact other routines that might run in a shared thread process.
5. If the routine is an external routine, you must put the routine implementation library or class file on the database server. Follow the general recommendations for deploying routines and the specific recommendations for deploying external routine library or class files.

Authorizations and binding of routines that contain SQL

To successfully invoke routines, you must have multiple authorizations and bindings 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** or **BIND** command parameter 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
- DATAACCESS 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 any user with ACCESSCTRL or SECADM authority, CONTROL or EXECUTE WITH GRANT OPTION privilege on the package. (The creator of a package automatically receives CONTROL and EXECUTE WITH GRANT OPTION on the package.)

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** or **BIND** command parameter 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), ACCESSCTRL, or SECADM 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** or **BIND** command parameter. In this case, the invoker of the routine assumes this value rather than the ID of the user who is executing the application.

Note:

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:

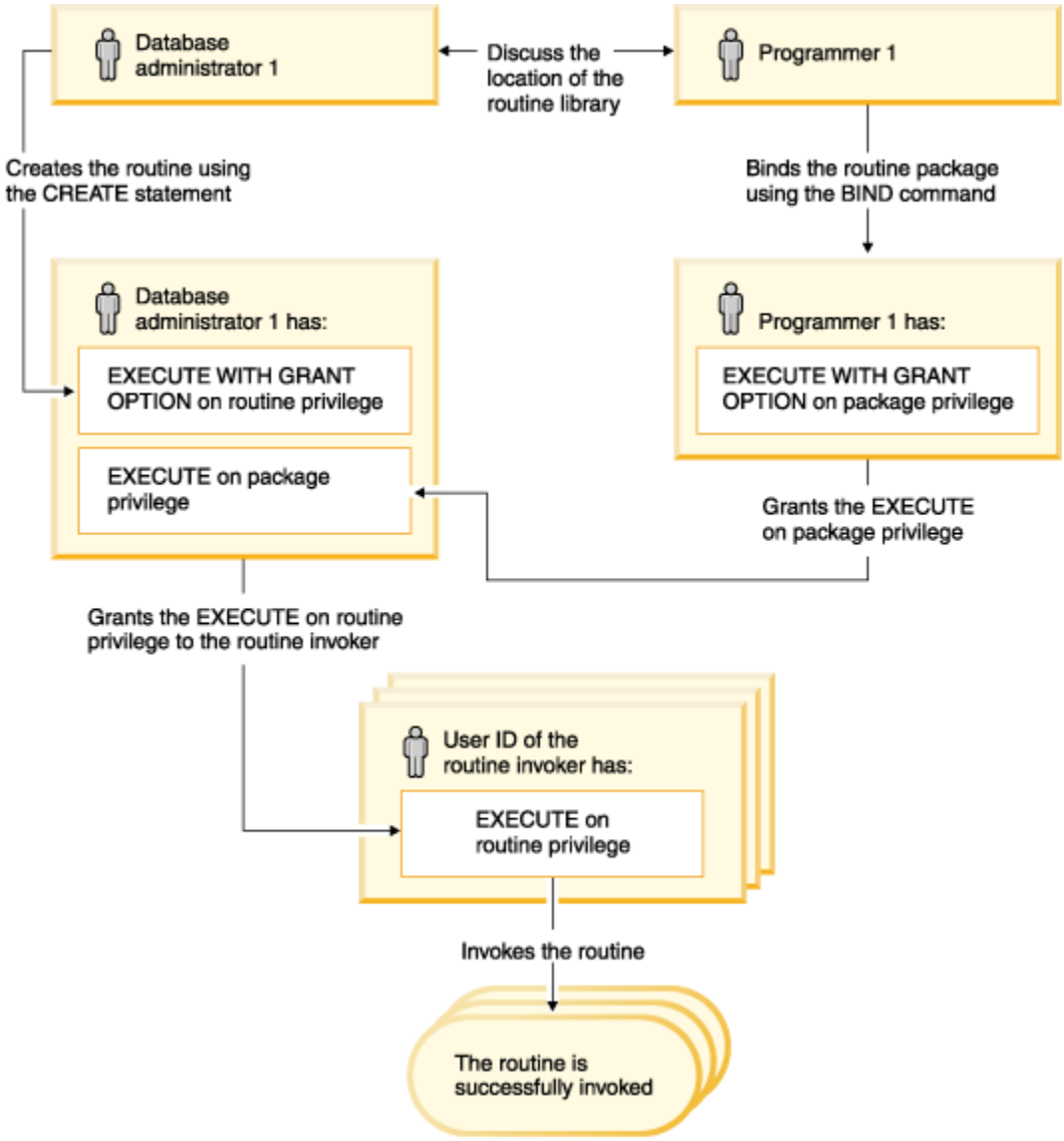


Figure 2. Managing the EXECUTE privilege on routines

1. Definer performs the appropriate CREATE statement to register the routine. This registers the routine in the database 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, the database manager 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).

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, the database manager do not allow operations that conflict on any table.

To describe how the database manager avoid 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 (because 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 because 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.

Debugging compiled SQL PL and PL/SQL objects overview

Debugging compiled SQL PL and PL/SQL objects can affect database security.

Within a debug session, the debugging user can freely alter the values of local routine and global variables. With this freedom, the debugging user can change what row and column access control rules evaluate to. These changes might allow the user to access data that they are not authorized for.

To prevent variable changes that allow unauthorized access, only users who are members of the built-in role, SYSDEBUG, can debug compiled SQL PL and PL/SQL objects. The security administrator (SECADM) is the only authority that can grant or revoke membership to SYSDEBUG. This new role is meant to be used on a test system and not on a production system.

The following guidelines should be considered when working with SYSDEBUG:

- Grant membership only to users who have a specific need to perform debugging on compiled SQL PL and PL/SQL objects.
- Revoke membership immediately when the need for debugging is no longer required.
- Grant membership only in databases used for development or testing. Membership should not be granted within a production database environment.
- Create and enable an audit policy for the SYSDEBUG role.

External routines

External routines are routines that have their logic implemented in a programming language application that resides outside of the database, in the file system of the database server.

The association of the routine with the external code application is asserted by the specification of the EXTERNAL clause in the CREATE statement of the routine.

You can create external procedures, external functions, and external methods. Although they are all implemented in external programming languages, each routine functional type has different features. Before deciding to implement an external routine, it is important that you first understand what external

routines are, and how they are implemented and used, by reading the topic, "Overview of external routines". With that knowledge you can then learn more about external routines from the topics targeted by the related links so that you can make informed decisions about when and how to use them in your database environment.

External routine features

External routines provide support for most of the common routine features and support for additional features that are not supported by SQL routines.

The following features are unique to external routines:

Access to files, data, and applications that resides outside of the database
External routines can access and manipulate data or files that reside outside of the database itself. They can also call applications that reside outside of the database. The data, files, or applications might, for example, reside in the database server file system or within the available network.

Variety of external routine parameter style options
The implementation of external routines in a programming language can be done using a choice of parameter styles. Although there might be a preferred parameter style for a chosen programming language, there is sometimes choice. Some parameter styles support passing of additional database and routine property information to and from the routine with the `dbinfo` structure.

Preservation of state between external function invocations with a scratchpad
External user-defined functions provide support for state preservation between function invocations for a set of values. The state preservation is done with a structure called a `scratchpad`. The state preservation can be useful for functions that return aggregated values and functions that require initial setup logic such as initialization of buffers.

Call-types identify individual external function invocations
External user-defined functions are invoked multiple times for a set of values. Each invocation is identified with a call-type value that can be referenced within the function logic. For example, there are special call-types for the following function calls:

- First invocation of a function
- Data fetching calls
- Final invocation

Call-types are useful because specific logic can be associated with a particular call-type.

External function and method features
External functions and external methods provide support for functions that, for a given set of input data, might be invoked multiple times and produce a set of output values.

To learn more about the features of external functions and methods, see the following topics:

- “External scalar functions” on page 56
- “External scalar function and method processing model” on page 57
- “External table functions” on page 57
- “External table function processing model” on page 58
- “Table function execution model for Java” on page 60
- “Scratchpads for external functions and methods” on page 61
- “Scratchpads on 32-bit and 64-bit operating systems” on page 64

These features are unique to external functions and methods and do not apply to SQL functions and SQL methods.

External scalar functions
External scalar functions are scalar functions that have their logic implemented in an external programming language.

These functions can be developed and used to extend the set of existing SQL functions and can be invoked in the same manner as database built-in functions such as LENGTH and COUNT. That is, they can be referenced in SQL statements wherever an expression is valid.

The execution of external scalar function logic takes place on the database server, however unlike built-in or user-defined SQL scalar functions, the logic of external functions can access the database server filesystem, perform system calls or access a network.

External scalar functions can read SQL data, but cannot modify SQL data.

External scalar functions 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 using the scratchpad to store some values that can be accessed or updated in subsequent invocations of the scalar function.

Features of external scalar functions

- Can be referenced as part of an SQL statement anywhere an expression is supported.
- The output of a scalar function 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.

Limitations

- Cannot do transaction management within a scalar function. That is, you cannot issue a COMMIT or a ROLLBACK within a scalar function.
- Cannot return result sets.
- Scalar functions are intended to return a single scalar value per set of inputs.
- External scalar functions are not intended to be used for a single invocation. They are designed such that for a single reference to the function and a given set of inputs, that the function be invoked once per input, and return a single scalar value. On the first invocation, scalar functions can be designed to do some setup work, or store some information that can be accessed in subsequent invocations. SQL scalar functions are better suited to functionality that requires a single invocation.
- In a single partition database external scalar functions 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 function. SQL scalar functions can contain SQL statements that read or modify data.

Common uses

- Extend the set of database 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

- C
- C++
- Java
- OLE

- .NET common language runtime languages

Note:

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 function, AVG(SHOESIZE), as an aggregate function sourced on the existing built-in aggregate function, AVG(INTEGER).
2. You can also create function that return a row. These are known as row functions and can only be used as a transform function for structured types. The output of a row function is a single row.

External scalar function and method processing model

The FIRST call, NORMAL call, and FINAL call are processing models for methods and scalar UDFs that are defined with the FINAL CALL specifications.

Use the following calls for ordinary error processing for methods and scalar UDFs.

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, the database manager cannot make the indicated calls.

External 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 the database 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 database 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 the database 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. The database 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.

External table function processing model

The processing model for external table functions consists of different calls to ensure that table user-defined functions (UDFs) are processed and executed correctly.

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.

Generic table functions

A generic table function is a table UDF where the output table is not specified when the UDF is defined. Instead, the output table is specified when the UDF is referenced. Different output table size and shape are possible for the same generic table function, depending on different input arguments.

You can create generic table functions with the Java programming language.

To define a generic table function, use the CREATE FUNCTION statement, and specify the RETURNS GENERIC TABLE option. To use this option, you must specify the LANGUAGE JAVA and PARAMETER STYLE DB2GENERAL options.

In the following example, the names and types of output columns are not specified:

```
CREATE FUNCTION csvRead (VARCHAR(255))
RETURNS GENERIC TABLE
EXTERNAL NAME 'UDFcsvReader!csvReadString'
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
```

Once the function is defined, you can access the function output using an SQL select statement, which includes a typed correlation clause. In the following example, the SELECT statement is used to indicate that the table contains two columns: USER, which is an INTEGER data type, and LINK, which is a VARCHAR(100) data type.

```
Select TX.*
From TABLE (csvRead('/TMP/data/userWebClicks.log'))
     AS TX (USER INTEGER, LINK VARCHAR(100))
WHERE TX.LINK LIKE 'www.ibm.com%'
```

You can use another SELECT statement to access the output from the same generic table function. In the following example, the SELECT statement is used to indicate that this time the table contains three different columns: CUSTOMERID, which is an INTEGER data type; NAME, which is a VARCHAR(100) data type; and ADDRESS, which is a VARCHAR(100) data type.

```
Select TX.*
From TABLE (csvRead('/TMP/data/customerWebClicks.log'))
     AS TX (CUSTOMERID INTEGER, NAME VARCHAR(100), ADDRESS VARCHAR(100))'
```

Using generic table functions

Use generic table functions when you have to produce different customized result sets based on different unstructured data inputs. For example, you might have data stored in CSV (comma-separated values) files, with different schemas.

About this task

If you want to join data that is stored in relational tables in a database with data stored in a CSV file, then you must use a generic Java table function to read and parse the data.

Procedure

1. Create a Java class that takes the CSV file handle as input, reads the file and returns rows of data to the database.

   A sample Java file with a simple CSV reader is provided in the following location:

   - For UNIX or Linux: `sqllib/samples/java/jdbc/UDFcsvReader.java`
   - For Windows: `sqllib\samples\java\jdbc\UDFcsvReader.java`
2. Compile the external Java routine.
3. Copy the class file to the following location: `$INSTALLDIR/sqllib/function`.
4. Register the Java function in the database.

```
CREATE FUNCTION CSVREAD
                (
                  filename VARCHAR(255)
                )
                RETURNS GENERIC TABLE
                EXTERNAL NAME 'UDFcsvReader!csvReadString'
                LANGUAGE JAVA
                PARAMETER STYLE DB2GENERAL
                VARIANT
                FENCED THREADSAFE
                DISALLOW PARALLEL
                NO DBINFO;
```

What to do next

You can now use the new Java function to analyze your data. For example, issue the following SELECT command:

```
SELECT  csv.name, csv.department, csv.id
    from T1, TABLE (CSVREAD( '~/csvfiles/file_1'))
    AS csv (name varchar(128), department varchar(128), id int)
    where T1.ID = csv.id;
```

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 during the processing of a statement by the database manager.

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.)

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
At each FETCH for a new row of table function data	• UDF method is called with FETCH call. • Method fetches and returns next row of data, or EOT.	• UDF method is called with FETCH call. • Method fetches and returns new row of data, or EOT.
At each CLOSE of the table function	• UDF method is called with CLOSE call. `close()` method if it exists for class. • Method closes its Web scan and disconnects from the Web server. `close()` does not need to do anything.	• UDF method is called with CLOSE call. • Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist.
After the last CLOSE of the table function	• No calls.	• UDF method is called with FINAL call. `close()` method is called if it exists for class. • Method disconnects from the Web server. `close()` method does not need to do anything.

Note:

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.

Scratchpads for external functions 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 the database manager. 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 the database manager 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. The database manager 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), the database manager refreshes the scratchpad before each OPEN call. If you specify the table function option FINAL CALL, the database manager does not examine or change the content of the scratchpad after its initialization. For scalar functions defined with scratchpads, the database manager 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 the database manager 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 database agent (an agent is a database 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 the database manager 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 the database manager 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 database crash.

As 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, the database manager 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, note that your UDF or method receives a call of type FIRST. This could be used to acquire and initialize some persistent resource.

Following is a simple Java example of a UDF that uses a scratchpad to compute the sum of squares of entries in a column. This example takes in a column and returns a column containing the cumulative sum of squares from the top of the column to the current row entry:

```
CREATE FUNCTION SumOfSquares(INTEGER)
RETURNS INTEGER
EXTERNAL NAME 'UDFsrv!SumOfSquares'
DETERMINISTIC
NO EXTERNAL ACTION
FENCED
NOT NULL CALL
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
NO SQL
SCRATCHPAD 10
FINAL CALL
DISALLOW PARALLEL
NO DBINFO@


// Sum Of Squares using Scratchpad UDF
public void SumOfSquares(int inColumn,
                         int outSum)
throws Exception
{
  int sum = 0;
  byte[] scratchpad = getScratchpad();

  // variables to read from SCRATCHPAD area
  ByteArrayInputStream byteArrayIn = new ByteArrayInputStream(scratchpad);
  DataInputStream dataIn = new DataInputStream(byteArrayIn);

  // variables to write into SCRATCHPAD area
  byte[] byteArrayCounter;
  int i;
  ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream(10);
  DataOutputStream dataOut = new DataOutputStream(byteArrayOut);

  switch(getCallType())
  {
    case SQLUDF_FIRST_CALL:
      // initialize data
      sum = (inColumn * inColumn);
      // save data into SCRATCHPAD area
      dataOut.writeInt(sum);
      byteArrayCounter = byteArrayOut.toByteArray();
      for(i = 0; i < byteArrayCounter.length; i++)
      {
        scratchpad[i] = byteArrayCounter[i];
      }
      setScratchpad(scratchpad);
      break;
    case SQLUDF_NORMAL_CALL:
      // read data from SCRATCHPAD area
      sum = dataIn.readInt();
      // work with data
```

```
        sum = sum + (inColumn * inColumn);
        // save data into SCRATCHPAD area
        dataOut.writeInt(sum);
        byteArrayCounter = byteArrayOut.toByteArray();
        for(i = 0; i < byteArrayCounter.length; i++)
        {
          scratchpad[i] = byteArrayCounter[i];
        }
        setScratchpad(scratchpad);
        break;
    }
    //set the output value
    set(2, sum);
  } // SumOfSquares UDF
```

Please note that there is a built-in database function that performs the same task as the `SumOfSquares` UDF. This example was chosen to demonstrate the use of a scratchpad.

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 be cautious when 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.

Following is a sample structure declaration for a scratchpad:

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

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

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

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

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

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

As the application cannot change the value in the length field of the scratchpad, 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.

External routine parameter styles

External routine implementations must conform to a particular convention for the exchange of routine parameter values. These conventions are known as *parameter styles*.

An external routine parameter style is specified when the routine is created by specifying the PARAMETER STYLE clause. Parameter styles characterize the specification and order in which parameter values are passed to the external routine implementation. They also specify what if any additional values are passed to the external routine implementation. For example, some parameter styles require that an additional separate null-indicator value is specified for each routine parameter value to provide information about the nullability. The null-indicator value provides information about the nullability of a parameter that is not easily determined with the native programming language data type.

The following table provides a list of the available parameter styles, the routine implementations that support each parameter style, the functional routine types that support each parameter style, and a description of the parameter style:

Table 8. Parameter styles

Parameter style	Supported language	Supported routine type	Description
SQL "1" on page 67	• C/C++ • OLE • .NET common language runtime languages • COBOL "2" on page 67	• 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 that is declared in the CREATE statement. • The SQLSTATE to be returned to the database manager. • The qualified name of the routine. • The specific name of the routine. • The SQL diagnostic string to be returned to the database manager. Depending on options that are 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" on page 67	• 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 that contains a null indicator for each parameter on the CALL statement. • The SQLSTATE to be returned to the database manager. • The qualified name of the stored procedure. • The specific name of the stored procedure. • The SQL diagnostic string to be returned to the database manager. 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.

Table 8. Parameter styles (continued)

Parameter style	Supported language	Supported routine type	Description
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 are 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. PARAMETER STYLE JAVA routines do not support the DBINFO or PROGRAM TYPE clauses. For UDFs, PARAMETER STYLE JAVA can be specified only when no structured data types are specified as parameters. Also, you cannot specify the following data types for return types of PARAMETER STYLE JAVA UDFs: • Structured type • CLOB • DBCLOB • BLOB Also, PARAMETER STYLE JAVA UDFs do not support table functions, call types, or scratchpads.
DB2GENERAL	• Java	• UDFs • stored procedures • methods	This type of routine uses 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, 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 for z/OS.
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 that contains 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 for z/OS.

Note:

1. For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.
2. COBOL can be used only to develop stored procedures.
3. .NET common language runtime methods are not supported.

XML data type support in external routines

The programming languages C, C++, COBOL, .NET CLR, and Java support parameters and variables of data type XML.

XML data type values are represented in external routine code in the same way as CLOB data types.

When declaring external routine parameters of data type XML, the CREATE PROCEDURE and CREATE FUNCTION statements that will be used to create the routines in the database must specify that the XML data type is to be stored as a CLOB data type. The size of the CLOB value should be close to the size of the XML document represented by the XML parameter.

The following CREATE PROCEDURE statement shows a CREATE PROCEDURE statement for an external procedure implemented in the C programming language with an XML parameter named `parm1`:

```
CREATE PROCEDURE myproc(IN parm1 XML AS CLOB(2M), IN parm2 VARCHAR(32000))
LANGUAGE C
FENCED
PARAMETER STYLE SQL
EXTERNAL NAME 'mylib!myproc';
```

Similar considerations apply when creating external UDFs, as shown in the following example:

```
CREATE FUNCTION myfunc (IN parm1 XML AS CLOB(2M))
RETURNS SMALLINT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NOT FENCED
NULL CALL
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'mylib1!myfunc'
```

XML data is materialized when passed to stored procedures as IN, OUT, or INOUT parameters. If you are using Java stored procedures, the heap size (**`java_heap_sz`** configuration parameter) might need to be increased based on the quantity and size of XML arguments, and the number of external stored procedures that are being executed concurrently.

Within external routine code, XML parameter and variable values are accessed, set, and modified in the same way as in database applications.

Restrictions on external routines

When you develop or debug external routines, you must be aware of certain restrictions on these routines.

Some restrictions on external routines apply to all external routines, while some apply only to external procedures or external functions. The following lists describe the restrictions.

Restrictions that apply to all external routines:

- New threads and processes cannot be created in external routines.
- Connection level APIs cannot be called from within external functions or external methods.
- Receiving inputs from the keyboard and displaying outputs to standard output is not possible from external routines. Do not use standard input-output streams. For example:
  - In external Java routine code, do not issue the `System.out.println()` methods.
  - In external C or C++ routine code, do not issue `printf()`.

  - In external COBOL routine code, do not issue `display`

    Although external routines cannot display data to standard output, they can include code that writes data to a file on the database server file system.

    For fenced routines that run in Linux or UNIX environments, 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: The database manager does not attempt to synchronize any external input or output activities that are performed by a routine with the database 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.
- Connection-related statements or commands cannot be executed in external routines. This restriction applies to the following statements and commands:
  - **BACKUP DATABASE**
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - **CREATE DATABASE**
  - **DROP DATABASE**
  - FORWARD RECOVERY
  - **RESTORE DATABASE**
- Avoid the use of operating system functions within routines. The use of operating system functions is restricted in the following cases:
  - User-defined signal handlers must not be installed for external routines. Failure to adhere to this restriction can result in unexpected external routine run-time failures, database abends, or other problems. Installing signal handlers can also interfere with operation of the JVM for Java routines.
  - System calls that terminate a process can abnormally terminate one of the database processes and result in database system or database application failure.

    Other system calls can also cause problems if they interfere with the normal operation of the database manager. For example, a function that attempts to unload a library containing a user-defined function from memory could cause severe problems. Be careful in coding and testing external routines containing system calls.
- Use of operating system function that results in the creation of a new process cannot be used in unfenced routines. These functions include `fork()`, `popen()`, and `system()`. Using these functions can interfere with the communications between the database servers and the cluster caching facility, which causes the routine to return an SQL0430N error.
- External routines must not contain commands that would terminate the current process. An external routine must always return control to the database manager without terminating the current process.
- External routine libraries, classes, or assemblies must not be updated while the database is active except in special cases. If an update is required while the database manager is active, and stopping and starting the instance is not an option, create the new library, class, or assembly for the routine with a different. Then, use the ALTER statement to change the external routine's EXTERNAL NAME clause value so that it references the name of the new library, class, or assembly file.
- Environment variable **DB2CKPTR** is not available in external routines. All other environment variables with names beginning with 'DB' are captured at the time the database manager is started and are available for use in external routines.
- Some environment variables with names that do not start with 'DB' are not available to external routines that are fenced. For example, the **LIBPATH** environment variable is not available for use. However these variables are available to external routines that are not fenced.

- Environment variable values that were set after the database manager is started are not available to external routines.
- Use of protected resources, resources that can only be accessed by one process at a time, within external routines should be limited. If used, try to reduce the likelihood of deadlocks when two external routines try to access the protected resource. If two or more external routines deadlock while attempting to access the protected resource, the database manager will not be able to detect or resolve the situation. This will result in hung external routine processes.
- Memory for external routine parameters should not be explicitly allocated on the database server. The database manager automatically allocates storage based upon the parameter declaration in the CREATE statement for the routine. Do not alter any storage pointers for parameters in external routines. 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 external routines. The database manager cannot guarantee that the memory used by static or global variables will be untouched between external routine invocations. For UDFs and methods, you can use scratchpads to store values for use between invocations.
- All SQL parameter values are buffered. This means that a copy of the value is made and passed to the external routine. If there are changes made to the input parameters of an external routine, these changes will have no effect on SQL values or processing. However, if an external 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.
- The **LOAD** utility does not support loading into tables that contain columns that reference fenced procedures. If you issue the **LOAD** command on such table, you will receive error message SQL1376N. To work around this restriction, you can redefine the routine to be unfenced, or use the import utility.
- Modifications to the Java Virtual Machine (JVM) or the JVM start arguments is not supported for external Java routines.

Restrictions that apply to external procedures only

- 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.

Restrictions that apply to external functions only

- External functions cannot return result sets. All cursors opened within an external function must be closed by the time the final-call invocation of the function completes.
- Dynamic allocations of memory in an external routine should be freed before the external routine returns. Failure to do so will result in a memory leak and the continuous growth in memory consumption of a database process that could result in the database system running out of memory.

  For external user-defined functions and external methods, scratchpads can be used to allocate dynamic memory required for multiple function invocations. When scratchpads are used in this way, specify the FINAL CALL attribute in the CREATE FUNCTION or CREATE METHOD statement. This ensures that allocated memory is freed before the routine returns.

Creating external routines

External routines including procedures and functions are created in a similar way as routines with other implementations, however there are a few more steps required, because the routine implementation requires the coding, compilation, and deployment of source code.

Before you begin

- The IBM Data Server Client must be installed.

- The database server must be running an operating system that supports the chosen implementation programming language compilers and development software.
- The required compilers and runtime support for the chosen programming language must be installed on the database server
- Authority to execute the CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statement.

Restrictions

For a list of restrictions that are associated with external routines see:

- "Restrictions on external routines" on page 67

About this task

You would choose to implement an external routine 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 programming language rather than using SQL and SQL PL statements.
- You require the routine logic to perform operations external to the database such as writing or reading to a file on the database server, the running of another application, or logic that cannot be represented with SQL and SQL PL statements.

Procedure

1. Code the routine logic in the chosen programming language.
   - For general information about external routines, routine features, and routine feature implementation, see the topics that are referenced in the Prerequisites section.
   - Use or import any header files that are required to support the execution of SQL statements.
   - Declare variables and parameters correctly using programming language data types that map to Db2 SQL data types.
2. Parameters must be declared in accordance with the format required by the parameter style for the chosen programming language. For more on parameters and prototype declarations see:
   - "External routine parameter styles" on page 65
3. Build your code into a library or class file.
4. Copy the library or class file into the *function directory* on the database server. You can store assemblies or libraries that are associated with 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.

   The assembly or library must be accessible on all physical hosts that are listed in the `db2nodes.cfg` file. If you copied the assembly or library to a directory that is not located on a shared file system, which is accessible to all hosts, then you must manually copy the file to the same location on each host.

   If your routine is written in Java, consider using the SQLJ.INSTALL_JAR built-in procedure to manage the deployment of the routine implementation.
5. 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 the appropriate value for the chosen API or programming language. Examples include: CLR, C, JAVA.
- 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 library, class, or assembly file to be associated with the routine using one of the following values:
  - the fully qualified path name of the routine library, class, or assembly file .
  - the relative path name of the routine library, class, or assembly file relative to the function directory.

  The database manager searches for the library, class, or assembly file by name in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.
- Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure that returns one or more result sets to the caller.
- Specify any other clauses that are required to characterize the routine.

What to do next

To call your external routine, see Routine invocation

Writing routines

The three types of routines (procedures, UDFs, and methods) have much in common with regards to how they are written.

Before you begin

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.

Before writing a routine, you must decide the following:

- The type of routine you need.
- The programming language you will use to write it.
- Which interface to use if you require SQL statements in your routine.

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.

Results

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.

External routine library and class management

To successfully develop and invoke external routines, external routine library and class files must be deployed and managed properly.

External routine library and class file management can be minimal if care is taken when external routines are first created and library and class files deployed.

The main external routine management considerations are the following:

- Deployment of external routine library and class files
- Security of external routine library and class files
- Resolution of external routine libraries and classes
- Modifications to external routine library and class files
- Backup and restore of external routine library and class files
- Verify all routine libraries are in the `sqllib/function` directory and that they are in the right library. Choose the member where you want to have the final version of the routine libraries. The library is the same library as the last member where the **db2iupdt** command was executed.

System administrators, database administrators and database application developers should all take responsibility to ensure that external routine library and class files are secure and correctly preserved during routine development and database administration tasks.

Deployment of external routine libraries and classes

Deployment of external routine libraries and classes refers to the copying of external routine libraries and classes to the database server once they have been built from source code.

External routine library, class, or assembly files must be copied into the `sqllib/function` directory in the home directory of the database instance owner , or a subdirectory of the function directory on the database server. To find out more about the function directory, see the description of the EXTERNAL clause for either of the following SQL statements: CREATE PROCEDURE or CREATE FUNCTION.

You can copy the external routine class, library, or assembly to other directory locations on the server, depending on the API and programming language that is used to implement the routine, however this is generally discouraged. If you copied the external routine class, library, or assembly to other directory locations on the server, you must fully qualify the path name and ensure that this value is used with the EXTERNAL NAME clause.

Library and class files can be copied to the database server file system using most generally available file transfer tools. Java routines can be copied from a client computer to a database server using special built-in procedures designed specifically for this purpose. See the topics on Java routines for more details.

The library or class file must be accessible on all physical hosts that are listed in the `db2nodes.cfg` file. If you copied the assembly or library to a directory that is not located on a shared file system which is accessible to all hosts, then you must manually copy the file to the same location on each host.

When you issue the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION, be sure to specify the appropriate clauses, paying particular attention to the EXTERNAL NAME clause.

- Specify the LANGUAGE clause with the appropriate value for the chosen API or programming language. Examples include: CLR, C, JAVA.
- 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 library, class, or assembly file to be associated with the routine using one of the following values:
  - the fully qualified path name of the routine library, class, or assembly file.
  - the relative path name of the routine library, class, or assembly file relative to the function directory.

The database manager searches for the library, class, or assembly file by name in the function directory unless a fully qualified or relative path name is specified in the EXTERNAL clause.

Security of external routine library or class files
External routine libraries are stored in the file system on the database server and are not backed up or protected in any way by the database manager. For routines to continue to successfully be invoked, it is imperative that the library associated with the routine continue to exist in the location specified in the EXTERNAL clause of the CREATE statement used to create the routine.

Do not move or delete routine libraries after creating routines; doing so will cause routine invocations to fail.

To prevent routine libraries from being accidentally or intentionally deleted or replaced, you must restrict access to the directories on the database server that contain routine libraries and restrict access to the routine library files. This can be done by using operating system commands to set directory and file permissions.

Resolution of external routine libraries and classes
The external routine library resolution is performed at the database instance level. In a database instance that contains multiple databases, external routine libraries of one database can be referenced with an EXTERNAL NAME clause of the CREATE statement in another database.

Instance level external routine resolution supports code re-use by allowing multiple routine definitions to be associated with a single library. When external routine libraries are not re-used in this way, and instead copies of the external routine library exist in the file system of the database server, library name conflicts can happen. This can specifically happen when there are multiple databases in a single instance and the routines in each database are associated with their own copies of 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).

To minimize the likelihood of this happening, it is recommended that a single copy of a routine library be stored in the instance level function directory (`sqllib/function` directory) and that the EXTERNAL clause of all of the routine definitions in each of the databases reference the unique library.

If two functionally different routine libraries must be created with the same name, it is important to take additional steps to minimize the likelihood of library name conflicts.

For C, C++, COBOL, and ADO.NET routines:
Library name conflicts can be minimized or resolved by:

1. Storing the libraries with routine bodies in separate directories for each database.
2. Creating the routines with an EXTERNAL NAME clause value that specifies the full path of the given library (instead of a relative path).

For Java routines:
Class name conflicts cannot be resolved by moving the class 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.

Modifications to external routine library and class files
Modifications to an existing external routine's logic might be necessary after an external routine has been deployed and it is in use in a production database system environment. Modifications to existing routines can be made, but it is important that they be done carefully so as to define a clear takeover point in time for the updates and to minimize the risk of interrupting any concurrent invocations of the routine.

If an external routine library requires an update, do not recompile and relink the routine to the same target file (for example, `sqllib/function/foo.a`) that 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 routine invocation to fail. If it is necessary to change the body of a routine without stopping and restarting the database manager, complete the following steps:

1. Create the new external routine library with a different library or class file name.
2. If it is an embedded SQL routine, bind the routine package to the database using the **BIND** command.
3. Use the ALTER ROUTINE statement to change the routine definition so that the EXTERNAL NAME clause references the updated routine library or class. 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.
4. For updating Java routines that are built into JAR files, you must issue a CALL SQLJ.REFRESH_CLASSES() statement to force the database manager to load the new classes. If you do not issue the CALL SQLJ.REFRESH_CLASSES() statement after you update Java routine classes, the database manager continues to use the previous versions of the classes. The database manager refreshes the classes when a COMMIT or ROLLBACK occurs.

Once the routine definition has been updated, all subsequent invocations of the routine will load and run the new external routine library or class.

Backup and restore of external routine library and class files
External routine libraries are not backed up with other database objects when a database backup is performed. They are similarly not restored when a database is restored.

If the purpose of a database backup and restore is to re-deploy a database, then external routine library files must be copied from the original database server file system to the target database server file system in such a way as to preserve the relative path names of the external routine libraries.

External routine library management and performance
External routine library management can affect the routine performance since the database manager dynamically caches external routine libraries to improve the routine performance.

For optimal external routine performance consider the following:

- Keep the number of routines in each library as small as possible. It is better to have numerous small external routine libraries than a few large ones.
- Group together within source code the routine functions of routines that are commonly invoked together. When the code is compiled into an external routine library the entry points of commonly invoked routines will be closer together which allows the database manager to provide better caching support. The improved caching support is due to the efficiency that can be gained by loading a single external routine library once and then invoking multiple external routine functions within that library.

  For external routines implemented in the C or C++ programming language, the cost of loading a library is paid only once for libraries that are consistently in use by C routines. After a routine is invoked once, all subsequent invocations from the same thread in the process, do not need to reload the routine's library.

.NET common language runtime (CLR) routines

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:

- “Benefits of using routines” on page 2
- “SQL data type representation in .NET CLR routines” on page 76
- “Parameters in .NET CLR routines” on page 78
- “Returning result sets from .NET CLR procedures” on page 80
- “Restrictions on .NET CLR routines” on page 82
- “Errors related to .NET CLR routines” on page 89

Developing a CLR routine is easy. For step-by-step instructions on how to develop a CLR routine and complete examples see:

- “Creating .NET CLR routines from the Db2 command window” on page 83
- “Examples of C# .NET CLR procedures” on page 91
- “Examples of C# .NET CLR functions” on page 114

Support for external routine development in .NET CLR languages
To develop external routines in .NET CLR languages and successfully run them, you need to use supported operating systems, versions of IBM database servers and clients, and development software.

.NET CLR external routines can be implemented in any language that can be translated into Microsoft intermediate language (MSIL). These languages include, but are not limited to: Managed C++, C#, and Visual Basic.

For list of supported Windows operating system, see the Windows operating system listing in Detailed system requirements for a specific product.

A supported version of the Microsoft .NET Framework software must also be installed on the same computer as the IBM database server or IBM data server client product. The Microsoft .NET Framework is independently available from the Microsoft site.

Tools for developing .NET CLR routines
Tools can make the task of developing .NET CLR routines that interact with a database faster and easier.

.NET CLR routines can be developed in the Microsoft Visual Studio environment with the following tool:

- IBM Database Add-Ins for Microsoft Visual Studio

The following command-line interfaces are available for developing .NET CLR routines on the database server:

- Db2Command Line Processor (CLP)
- Db2Command Window

Designing .NET CLR routines
When designing .NET CLR routines, you should take into account both general external routine design considerations and .NET CLR specific design considerations.

Knowledge and experience with .NET application development and general knowledge of external routines. The following topics can provide you with some of the required prerequisite information.

For more information on the features and uses of external routines see:

- "External routine implementation" on page 18

For more information on the characteristics of .NET CLR routines, see:

- ".NET common language runtime (CLR) routines" on page 75

With the prerequisite knowledge, designing embedded SQL routines consists mainly of learning about the unique features and characteristics of .NET CLR routines:

- Include assemblies that provide support for SQL statement execution in .NET CLR routines (`IBM.Data.DB2`)
- Supported SQL data types in .NET CLR routines
- Parameters to .NET CLR routines
- Returning result sets from .NET CLR routines
- Security and execution control mode settings for .NET CLR routines
- Restrictions on .NET CLR routines
- Returning result sets from .NET CLR procedures

After having learned about the .NET CLR characteristics, see: "Creating .NET CLR routines" on page 83.

SQL data type representation in .NET CLR routines
.NET CLR routines can reference SQL data type values as routine parameters, parameter values to be used as part of SQL statement execution, and as variables, however the appropriate IBM SQL data type values, IBM Data Server Provider for .NET data type values, and .NET Framework data type values must be used to ensure that there is no truncation or loss of data when accessing or retrieving the values.

For routine parameter specifications within the CREATE PROCEDURE or CREATE FUNCTION statements used to create .NET CLR routines, SQL data type values are used. Most SQL data types can be specified for routine parameters, however there are some exceptions.

For specifying parameter values to be used as part of an SQL statement to be executed, IBM Data Server Provider for .NET objects must be used. The `DB2Parameter` object is used to represent a parameter to be added to a `DB2Command` object which represents a SQL statement. When specifying the data type value for the parameter, the IBM Data Server Provider for .NET data type values available in the

IBM.Data.DB2Types namespace must be used. The IBM.Data.DB2Types namespace provides classes and structures to represent each of the supported SQL data types.

For parameters and local variables that might temporarily hold SQL data type values appropriate IBM Data Server Provider for .NET data types, as defined in the IBM.Data.DB2Types Namespace, must be used.

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

The following table shows mappings between DB2Type data types, database data types, Informix® data types, Microsoft .NET Framework types, and DB2Types classes and structures.

Category	DB2Types Classes and Structures	DB2Type Data Type	Database Data Type	Informix Data Type	.NET Data Type
Numeric	DB2Int16	SmallInt	SMALLINT	BOOLEAN, SMALLINT	Int16
Numeric	DB2Int32	Integer	INT	INTEGER, INT, SERIAL	Int32
Numeric	DB2Int64	BigInt	BIGINT	BIGINT, BIGSERIAL, INT8, SERIAL8	Int64
Numeric	DB2Real, DB2Real370	Real	REAL	REAL, SMALLFLOAT	Single
Numeric	DB2Double	Double	DOUBLE PRECISION	DECIMAL (≤31), DOUBLE PRECISION	Double
Numeric	DB2Double	Float	FLOAT	DECIMAL (32), FLOAT	Double
Numeric	DB2Decimal	Decimal	DECIMAL	MONEY	Decimal
Numeric	DB2DecimalFloat	DecimalFloat	DECFLOAT (16|34)[1,4]		Decimal
Numeric	DB2Decimal	Numeric	DECIMAL	DECIMAL (≤31), NUMERIC	Decimal
Date/Time	DB2Date	Date	DATE	DATETIME (date precision)	Datetime
Date/Time	DB2Time	Time	TIME	DATETIME (time precision)	TimeSpan
Date/Time	DB2TimeStamp	Timestamp	TIMESTAMP	DATETIME (time and date precision)	DateTime
XML	DB2Xml	Xml[2]	XML		Byte[]
Character data	DB2String	Char	CHAR	CHAR	String
Character data	DB2String	VarChar	VARCHAR	VARCHAR	String
Character data	DB2String	LongVarChar[1]	LONG VARCHAR	LVARCHAR	String
Binary data	DB2Binary	Binary	CHAR FOR BIT DATA		Byte[]
Binary data	DB2Binary	Binary[3]	BINARY		Byte[]

[1] These data types are not supported as parameters in the .NET common language runtime routines.
[2] A DB2ParameterClass.ParameterName property of the type DB2Type.Xml can accept variables of the following types: String, byte[], DB2Xml, and XmlReader.
[3] These data types are applicable only to Db2 for z/OS.
[4] This data type is only supported for Db2 for z/OS, Version 9 and later releases and for Db2 Version 9.5 and later releases.

Category	DB2Types Classes and Structures	DB2Type Data Type	Database Data Type	Informix Data Type	.NET Data Type
Binary data	DB2Binary	VarBinary[3]	VARBINARY		Byte[]
Binary data	DB2Binary	LongVarBinary[1]	LONG VARCHAR FOR BIT DATA		Byte[]
Graphic data	DB2String	Graphic	GRAPHIC		String
Graphic data	DB2String	VarGraphic	VARGRAPHIC		String
Graphic data	DB2String	LongVarGraphic[1]	LONG VARGRAPHIC		String
LOB data	DB2Clob	Clob	CLOB	CLOB, TEXT	String
LOB data	DB2Blob	Blob	BLOB	BLOB, BYTE	Byte[]
LOB data	DB2Clob	DbClob	DBCLOB		String
Row ID	DB2RowId	RowId	ROWID		Byte[]

Parameters in .NET CLR routines

Parameter declaration in .NET 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 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:

- "External routine parameter styles" on page 65

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 keyword 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 in the following section.

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

- "External routine parameter styles" on page 65

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` 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,
                                 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: The database manager 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 .NET CLR procedures” on page 80

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)
```

Returning result sets from .NET 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).

Before you begin

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.

An understanding of how to create CLR routines will help you to perform the steps in the following procedure for returning results from a CLR procedure.

- "Creating .NET CLR routines from the Db2 command window" on page 83

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.

Security and execution modes for CLR routines

As a database administrator or an application developer, you should protect the assemblies associated with your external routines from unwelcome tampering to restrict the actions of routines at run time.

.NET common language runtime (CLR) routines support the specification of an execution control mode that identifies what types of actions a routine will be allowed to perform at run time. At run time, the database manager can detect if the routine attempts to perform actions beyond the scope of its specified execution control mode, which can be helpful when determining whether an assembly has been compromised.

To set the execution control mode of a CLR routine, specify the optional EXECUTION CONTROL clause in the CREATE statement for the routine. Valid modes are:

- SAFE
- FILEREAD
- FILEWRITE
- NETWORK
- UNSAFE

To modify the execution control mode in an existing CLR routine, execute the ALTER PROCEDURE or ALTER FUNCTION statement.

If the EXECUTION CONTROL clause is not specified for a CLR routine, by default the CLR routine is run using the most restrictive execution control mode: SAFE. Routines that are created with this execution control mode can only access resources that are controlled by the database manager. Less restrictive execution control modes allow a routine to access files (FILEREAD or FILEWRITE) or perform network operations such as accessing a web page (NETWORK). The execution control mode UNSAFE specifies that no restrictions are to be placed on the behavior of the routine. Routines defined with UNSAFE execution control mode can execute binary code.

These modes represent a hierarchy of allowable actions, and a higher-level mode includes the actions that are allowed below it in the hierarchy. For example, execution control mode NETWORK allows a routine to access web pages on the internet, read and write to files, and access resources that are controlled by the database manager. It is recommended to use the most restrictive execution control mode possible, and to avoid using the UNSAFE mode.

If the database manager detects at run time that a CLR routine is attempting an action outside of the scope of its execution control mode, the database manager returns an error (SQLSTATE 38501).

The EXECUTION CONTROL clause can only be specified for LANGUAGE CLR routines. The scope of applicability of the EXECUTION CONTROL clause is limited to the .NET CLR routine itself, and does not extend to any other routines that it might call.

Refer to the syntax of the CREATE statement for the appropriate routine type for a full description of the supported execution control modes.

Restrictions on .NET 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.

The following restrictions affect .NET CLR routines.

The CREATE METHOD statement with LANGUAGE CLR clause is not supported

You cannot create external methods for the database 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.

EXECUTION CONTROL clause restricts the logic contained in the routine

The EXECUTION CONTROL clause and associated value determine what types of logic and operations can be executed in a .NET CLR routine. By default the EXECUTION CONTROL clause value is set to SAFE. For routine logic that reads files, writes to files, or that accesses the internet, a non-default and less restrictive value for the EXECUTION CONTROL clause must be specified.

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

The DECIMAL data type in the database is represented with a 31-digit precision and 28-digit scale. The .NET CLR `System.Decimal` data type is limited to a 29-digit precision and 28-digit scale. Therefore, external CLR routines must not assign a value to a `System.Decimal` data type that has a value greater than `(2^96)-1`, which is the highest value that can be represented using a 29-digit precision and 28-digit scale. The database manager returns a runtime error (SQLSTATE 22003, SQLCODE -413) if such an assignment occurs. At the time of execution of the CREATE statement for the routine, if a DECIMAL data type parameter is defined with a scale greater than 28, the database manager returns an error (SQLSTATE 42613, SQLCODE -628).

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

Data types not supported in CLR routines

The following SQL data types are not supported in CLR routines:

- BINARY
- VARBINARY
- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- LONG GRAPHIC
- 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.

.NET CLR not supported for implementing security plug-ins

The .NET CLR is not supported for compiling and linking source code for security plug-in libraries.

Creating .NET CLR routines

Creating .NET CLR routines consists of executing a CREATE statement that defines the routine in a database server, and developing the routine implementation that corresponds to the routine definition.

Before you begin

- Review the ".NET common language runtime (CLR) routines" on page 75.
- Ensure that you have access to a database server, including instances and databases.
- Ensure that the operating system is at a version level that is supported by the database products.
- Ensure that the Microsoft .NET development software is at a version level that is supported for .NET CLR routine development. Refer to "Support for external routine development in .NET CLR languages" on page 75.
- Authority to execute the CREATE PROCEDURE or CREATE FUNCTION statement.

For a list of restrictions that are associated with CLR routines see:

- "Restrictions on .NET CLR routines" on page 82

About this task

The ways in which you can create .NET CLR routines follow:

- Using the graphical tool that is provided with the IBM Database Add-Ins for Microsoft Visual Studio
- Using the Db2 command window

It is easier to create .NET CLR routines using the IBM Database Add-Ins for Microsoft Visual Studio. However, you can use the Db2 command window to create .NET CLR routines.

Create .NET CLR routines from one of the following interfaces:

Procedure

- Visual Studio .NET when the IBM Database Add-Ins for Microsoft Visual Studio is also installed. When the Add-In is installed, graphical tool that is integrated into Visual Studio .NET is available for creating .NET CLR routines.
- Db2 command window

What to do next

To create .NET CLR routines from Db2 command window, see:

- "Creating .NET CLR routines from the Db2 command window" on page 83

Creating .NET CLR routines from the Db2 command window

Procedures and functions that reference an intermediate language assembly are created in the same way as any external routine is created.

Before you begin

- Knowledge of CLR routine implementation. To learn about CLR routines in general and about CLR features, see:
  - ".NET common language runtime (CLR) routines" on page 75
- The database server must be running a Windows operating system that supports the Microsoft .NET Framework.

- A supported version of the Microsoft .NET Framework software must be installed on the server. The .NET Framework is independently available or as part of the Microsoft .NET Framework Software Development Kit.
- A supported database product or IBM Data Server Client must be installed. See the installation requirements for database products.
- 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 .NET CLR routines" on page 82

About this task

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.

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 "Before you begin" section
   - Use or import the `IBM.Data.DB2` assembly if your routine will execute SQL.
   - Declare host variables and parameters correctly using data types that map to SQL data types. For a data type mapping between the database and .NET data types, see the following topic:
     - "SQL data type representation in .NET CLR routines" on page 76
   - Parameters and parameter null indicators must be declared with one of the supported parameter styles and according to the parameter requirements for .NET CLR routines. Also, scratchpads for UDFs, and the DBINFO class are passed into CLR routines as parameters. For more on parameters and prototype declarations, see the following topic:
     - "Parameters in .NET CLR routines" on page 78
   - 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 .NET CLR procedures" on page 80
   - 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 the databases, see the following topic:
   - "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*
3. Copy the assembly into the database *function directory* on the database server. You can store assemblies or libraries that are associated with database routines in the function directory. For more information about the function directory, see the EXTERNAL clause of the CREATE PROCEDURE or CREATE FUNCTION statement.

You can copy the assembly to another directory on the server if you want, 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 the database manager 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 the database manager, you will receive an error (SQLCODE -20282) with reason code 1.
   - Specify the DYNAMIC RESULT SETS clause with an integer value equivalent to the maximum number of result sets that might be returned by the routine.
   - You can not specify the NOT FENCED clause for CLR procedures. By default CLR procedures are executed as FENCED procedures.

Building .NET CLR routine code

Once .NET CLR routine implementation code is written, it must be compiled and deployed before the routine can be called. The steps that are required to build .NET CLR routines are similar to steps for building other external routines.

Procedure

- There are three ways to build .NET CLR routines:
  - Using the graphical tool that is provided with the IBM Database Add-Ins for Microsoft Visual Studio
  - Using sample batch files
  - Entering commands from the Db2 command window

  The sample build scripts and batch files for routines are designed for building database sample routines (procedures and user-defined functions) as well as user created routines for a particular operating system using the default supported compilers.

  There is a separate set of sample build scripts and batch files for C# and Visual Basic. It is easier to build .NET CLR routines using the graphical tools or the build scripts, which can be modified as required. However, it is often helpful to know how to build routines from the Db2 command window.

Building .NET common language runtime (CLR) routine code using sample build scripts

Building .NET common language runtime (CLR) routine source code is a subtask of creating .NET CLR routines. You can use the sample batch files that are provided with the database to build .NET CLR routines more easily.

The sample build scripts can be used for source code with or without SQL statements. The build scripts take care of the compilation, linking, and deployment of the built assembly to the `function directory`.

As alternatives, you can build .NET CLR routines in Visual Studio .NET to simplify the task or you can issue the commands in the sample build script manually.

You can use the programming-language specific sample build script **`bldrtn`** for building C# and Visual Basic .NET CLR routines. They are located in the database sample directory along with sample programs:

- For C: `sqllib/samples/cs/`
- For C++: `sqllib/samples/vb/`

The **bldrtn** scripts can be used to build source code files containing both procedures and user-defined functions. The script does the following:

- Establishes a connection with a user-specified database
- Compiles and links the source code to generate an assembly with a .DLL file suffix
- Copies the assembly file to the database function directory on the database server

The **bldrtn** scripts accept two arguments:

- The name of a source code file without any file suffix
- The name of a database to which a connection will be established

The database parameter is optional. If no database name is supplied, the program uses the default sample database. As routines must be built on the same instance where the database resides, no arguments are required for a user ID and password.

Prerequisites

- The required .NET CLR routine operating system and development software prerequisites must be satisfied. See: "Support for .NET CLR routine development".
- Source code file containing one or more routine implementations.
- The name of the database within the current database instance in which the routines are to be created.

Procedure

To build a source code file that contains one or more routine code implementations, perform the following steps.

1. Open the Db2 command window.
2. Copy your source code file into the same directory as the **bldrtn** script file.
3. If the routines will be created in the sample database, enter the build script name followed by the name of the source code file without the .cs or .vb file extension:

   ```
   bldrtn file-name
   ```

   If the routines will be created in another database, enter the build script name, the source code file name without any file extension, and the database name:

   ```
   bldrtn file-name database-name
   ```

   The script compiles and links the source code and produces an assembly. The script then copies the assembly to the function directory on the database server
4. If this is not the first time that the source code file containing the routine implementations was built, stop and restart the database to ensure the new version of the shared library is used by the database. You can do this by entering **db2stop** followed by **db2start** on the command line.

Once you have successfully built the routine shared library and deployed it to the function directory on the database server, you should complete the steps associated with the task of creating C and C++ routines.

Creating .NET CLR routines includes a step for executing the CREATE statement for each routine that was implemented in the source code file. After routine creation is completed you can invoke your routines.

Building .NET common language runtime (CLR) routine code from the Db2 command window
You can build .NET CLR routine from the command line.

Before you begin

As an alternative to manually building the .NET CLR routines, you can use the sample build scripts that are provided in the `sample` subdirectory of the `sqllib` directory or build the routine in the Visual Studio .NET project.

The following conditions must be met before you can build the .NET CLR routines.

- Prerequisite requirements for the operating system and the .NET CLR routine development software must be met.
- You must have a .NET CLR routine source code that is written in a supported .NET CLR programming language.
- You must have the database information in which the routines are to be created.
- You must have compiler options that are operating system specific and link options that are required for building .NET CLR routines.

Procedure

To build a source code file that contains one or more .NET CLR routine code implementations:

1. Open the Db2 command window.
2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database in which the routines are created.
4. Compile the source code file.
5. Link the source code file to generate a shared library. The linking process requires the use of compiler link options that references the database `include` directory.
6. Copy the assembly file with the .DLL file suffix to the database `function` directory on the database server.
7. If you are not building the source code file for the first time, you must stop and restart the database manager to ensure that the new version of the shared library is used by the database manager. You can issue the **db2stop** command followed by the **db2start** command to restart the database manager.

Results
Once you successfully built and deployed the routine library, you can register the new routine in the database by running the **CREATE** statement. The routine must be registered before you can call the new routine.

Example

The following example builds a .NET CLR source code file. Steps are shown for both the `myVBfile.vb` Visual Basic code file and the `myCSfile.cs` C# code file that contains routine implementations. The routines are being built with the Microsoft .NET Framework on a Windows operating system to generate a 64-bit assembly.

1. Open the Db2 command window.
2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database to which the routine is intended for.

```
Db2 connect to database-name
```

4. Compile the source code file and specify any compiler options that you need to specify. You must reference the IBM .NET data provider file location. In the following example. the $DB2PATH environment variable represents the IBM data server product installation path:

```
C# example
====================
csc /out:myCSfile.dll /target:library
    /reference:$DB2PATH%\bin\netf40\IBM.Data.DB2.dll myCSfile.cs

Visual Basic example
====================
vbc /target:library /libpath:$DB2PATH\bin\netf40
    /reference:$DB2PATH\bin\netf40\IBM.Data.DB2.dll
    /reference:System.dll
    /reference:System.Data.dll myVBfile.vb
```

   The compiler displays an output to indicate whether there are any errors.

5. Copy the shared library to the database `function` directory on the database server.

```
C# example
====================
rm -f ~HOME/sqllib/function/myCSfile.DLL
cp myCSfile $HOME/sqllib/function/myCSfile.DLL

Visual Basic example
====================
rm -f ~HOME/sqllib/function/myVBfile.DLL
cp myVBfile $HOME/sqllib/function/myVBfile.DLL
```

   This step ensures that the routine library is in the default directory where the database manager looks for routine libraries.

6. Stop and restart the database manager.

```
 db2stop
 db2start
```

You can modify the sample build scripts that are provided in the `sample` subdirectory of the `sqllib` directory to simplify building the .NET CLR routines.

Debugging .NET CLR routines

Debugging .NET CLR routines might be required if you fail to be able to create a routine, invoke a routine, or if upon invocation a routine does not behave or perform as expected.

About this task

Consider the following when debugging .NET CLR routines:

Procedure

- Verify that a supported operating system for .NET CLR routine development is being used.
- Verify that both database server and database client are supported for the .NET CLR routine development.
- Verify that supported Microsoft .NET Framework development software is being used.
- If routine creation failed:
  - Verify that the user has the required authority and privileges to execute the CREATE PROCEDURE or CREATE FUNCTION statement.
- If routine invocation failed:
  - Verify that the user has authority to execute the routine. If an error (SQLCODE -551, SQLSTATE 42501), this is likely because the invoker does not have the EXECUTE privilege on the routine. This privilege can be granted by any user with SECADM authority, ACCESSCTRL authority, or by any user with EXECUTE WITH GRANT OPTION privilege on the routine.

  - Verify that the routine parameter signature used in the CREATE statement for the routine matches the routine parameter signature in the routine implementation.
  - Verify that the data types used in the routine implementation are compatible with the data types specified in the routine parameter signature in the CREATE statement.
  - Verify that in the routine implementation that the .NET CLR language specific keywords used to indicate the method by which the parameter must be passed (by value or by reference) are valid.
  - Verify that the value specified in the EXTERNAL clause in the CREATE PROCEDURE or CREATE FUNCTION statement matches the location where the .NET CLR assembly that contains the routine implementation is located on the file system of the computer where the database server is installed.
  - If the routine is a function, verify that all of the applicable call types have been programmed correctly in the routine implementation. This is particularly important if the routine was defined with the FINAL CALL clause.
- If the routine is not behaving as expected:
  - Modify your routine such that it outputs diagnostic information to a file located in a globally accessible directory. Output of diagnostic information to the screen is not possible from .NET CLR routines. Do not direct output to files in directories used by the database.
  - Debug your routine locally by writing a simple .NET application that invokes the routine entry point directly. For information on how to use debugging features in Microsoft Visual Studio .NET, consult the Microsoft Visual Studio .NET compiler documentation.

Results
For more information on common errors related to .NET CLR routine creation and invocation, see:

- "Errors related to .NET CLR routines" on page 89

Errors related to .NET CLR routines
Although external routines share a generally common implementation, there are some database errors, which might arise that are specific to common language runtime (CLR) routines.

The following list contains of most commonly encountered SQLCODEs by .NET CLR routines. Database errors that are 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.

The error message details the cause of the error and the action that the user can take to resolve the error. Additional routine error scenario information can be found in the database diagnostic log file (`db2diag.log`).

CLR routine creation time errors

SQLCODE -451, SQLSTATE 42815
The -451 error is raised when the **CREATE TYPE** statement. which includes an external method that specifies the LANGUAGE clause with the CLR value is issued. You cannot create 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 quotation marks, object identifiers, and the separating characters (for example, `' <a> ! <b> '` is invalid). Path and file names, however, can 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 you use the short form file name for CLR routines on Linux or UNIX operating systems, the file must reside in the database function directory. If you use the short form file name for CLR routines on Windows operating systems, the file must reside in the system PATH environment variable. File extensions (examples: `.a` (on Linux and UNIX), `.dll` (on Windows)) must 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, the database manager presumes that the path name provided is a relative path name to the assembly, relative to the database 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. The database manager 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 database 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, ensure that the supported `.NET Framework` version is installed on the database server.

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. For errors of this type, the .NET stack traceback that indicates where the unhandled exception occurred will be written to the database diagnostic log file (`db2diag.log`).

This error can also occur if the routine attempted an action that is beyond the scope of allowed actions for the specified execution mode for the routine. In this case, an entry is made in the database diagnostic log file to indicate that the exception occurred due to an execution control violation. The exception stack traceback that indicates where the violation occurred is also logged.

Determine if the assembly of the routine has been compromised or recently modified. If the routine has been validly modified, this problem can be occurring because the EXECUTION CONTROL mode for the routine is no longer set to a mode that is appropriate for the changed logic. If you are certain that the assembly has not been wrongfully tampered with, you can modify the routine's execution mode with the ALTER PROCEDURE or ALTER FUNCTION statement as appropriate. Refer to the following topic for more information:

- "Security and execution modes for CLR routines" on page 81

Examples of .NET CLR routines
When developing .NET CLR routines, it is helpful to refer to examples to get a sense of what the CREATE statement and the .NET CLR routine code should look like.

About this task
The following topics contain examples of .NET CLR procedures and functions (including both scalar and table functions):

.NET CLR procedures

- Examples of Visual Basic .NET CLR procedures
- Examples of C# .NET CLR procedures

.NET CLR functions

- Examples of Visual Basic .NET CLR functions
- Examples of C# .NET CLR functions

Examples of C# .NET CLR procedures
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.

Before you begin

Before working with the CLR procedure examples you might want to read the following concept topics:

- ".NET common language runtime (CLR) routines" on page 75
- "Creating .NET CLR routines from the Db2 command window" on page 83
- Building common language runtime (CLR) .NET routines

About this task

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 C# .NET CLR functions" on page 114

The following examples 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" on page 92
- "Example 1: C# parameter style GENERAL procedure" on page 92
- "Example 2: C# parameter style GENERAL WITH NULLS procedure" on page 93
- "Example 3: C# parameter style SQL procedure" on page 94
- "Example 4: C# parameter style GENERAL procedure returning a result set" on page 95
- "Example 5: C# parameter style SQL procedure accessing the dbinfo structure" on page 96
- "Example 6: C# procedure with PROGRAM TYPE MAIN style" on page 97

Example

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:

```
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 the database manager 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.

```
CREATE PROCEDURE setEmpBonusGEN(IN empID CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
EXECUTION CONTROL SAFE
```

```
FENCED
THREADSAFE
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.

```
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
MODIFIES SQL DATA
EXECUTION CONTROL SAFE
FENCED
THREADSAFE
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
;
```

```
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.

```
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
MODIFIES SQL DATA
FENCED
THREADSAFE
EXECUTION CONTROL SAFE
```

```
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusSQL' ;
```

```
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.

```
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.

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

```
 public static void ReturnDbName(out string dbName,
                                 out Int16  dbNameNullInd,
                                 ref string sqlState,
                                     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 occurred"
    //
    //  the database manager returns the above values to the client in the
    //  SQLCA structure.  The values are used to generate a
```

```
    //  standard 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.

```
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
MODIFIES SQL DATA
FENCED
THREADSAFE
EXECUTION CONTROL SAFE
PROGRAM TYPE MAIN
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!main' ;
```

```
 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();
      }
   }
```

Examples of Visual Basic .NET CLR functions

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.

Before you begin

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

- ".NET common language runtime (CLR) routines" on page 75
- "Creating .NET CLR routines from the Db2 command window" on page 83
- "External scalar functions" on page 56
- Building common language runtime (CLR) .NET routines

About this task

For examples of CLR procedures in Visual Basic:

- "Examples of Visual Basic .NET CLR procedures" on page 102

The following examples 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" on page 98
- "Example 1: Visual Basic parameter style SQL table function" on page 99
- "Example 2: Visual Basic parameter style SQL scalar function" on page 100

Example

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:

```
  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 file system. 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.

```
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
EXECUTION CONTROL SAFE
```

```
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
```

```
 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.

```
  CREATE FUNCTION mycount(INTEGER)
  RETURNS INTEGER
  LANGUAGE CLR
  PARAMETER STYLE SQL
  NO SQL
  SCRATCHPAD 10
  FINAL CALL
  FENCED
  EXECUTION CONTROL SAFE
  NOT DETERMINISTIC
  EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp';
```

```
  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
```

Examples of Visual Basic .NET CLR procedures

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.

Before you begin

Before working with the CLR procedure examples you might want to read the following concept topics:

- ".NET common language runtime (CLR) routines" on page 75
- "Creating .NET CLR routines from the Db2 command window" on page 83
- "Benefits of using routines" on page 2
- Building common language runtime (CLR) .NET routines

About this task

For examples of CLR UDFs in `Visual Basic`:

- "Examples of Visual Basic .NET CLR functions" on page 98

The following examples 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" on page 102
- "Example 1: Visual Basic parameter style GENERAL procedure" on page 103
- "Example 2: Visual Basic parameter style GENERAL WITH NULLS procedure" on page 104
- "Example 3: Visual Basic parameter style SQL procedure" on page 105
- "Example 4: Visual Basic parameter style GENERAL procedure returning a result set" on page 106
- "Example 5: Visual Basic parameter style SQL procedure accessing the dbinfo structure" on page 106
- "Example 6: Visual Basic procedure with PROGRAM TYPE MAIN style" on page 107

Example

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:

```
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 the database manager 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.

```
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.

```
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.

```
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'
```

```
  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.

```
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.

```
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 occurred"
    '
    ' These will be returned by database managher in the SQLCA.
        ' It will appear in the format of a regular 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.

```
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'
```

```
  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
```

Example: XML and XQuery support in C# .NET CLR procedure

Once the basics of procedures, the essentials of .NET common language runtime routines, XQuery, and XML are understood, you can create and call CLR procedures with XML features.

The following example demonstrates a creation of C# .NET CLR procedure with parameters of type XML, which update and query XML data.

Prerequisites

Before you work with the CLR procedure example, you might want to read the following concept topics:

- .NET common language runtime (CLR) routines
- Creating .NET CLR routines from the Db2 command window
- Benefits of using routines

The examples use a table that is named `xmlDataTable`, which is created and populated with the following statements:

```
CREATE TABLE xmlDataTable
(
   num INTEGER,
   xdata XML
)

INSERT INTO xmlDataTable VALUES
   (1, XMLPARSE(DOCUMENT '<doc>
                              <type>car</type>
                              <make>Pontiac</make>
                              <model>Sunfire</model>
                              </doc>' PRESERVE WHITESPACE)),
   (2, XMLPARSE(DOCUMENT '<doc>
                              <type>car</type>
                              <make>Mazda</make>
                              <model>Miata</model>
```

```
                                </doc>' PRESERVE WHITESPACE)),
     (3, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Mary</name>
                                <town>Vancouver</town>
                                <street>Waterside</street>
                                </doc>' PRESERVE WHITESPACE)),
     (4, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Mark</name>
                                <town>Edmonton</town>
                                <street>Oak</street>
                                </doc>' PRESERVE WHITESPACE)),
     (5, XMLPARSE(DOCUMENT '<doc>
                                <type>animal</type>
                                <name>dog</name>
                                </doc>' PRESERVE WHITESPACE)),
     (6, NULL),
     (7, XMLPARSE(DOCUMENT '<doc>
                                <type>car</type>
                                <make>Ford</make>
                                <model>Taurus</model>
                                </doc>' PRESERVE WHITESPACE)),
     (8, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Kim</name>
                                <town>Toronto</town>
                                <street>Elm</street>
                                </doc>' PRESERVE WHITESPACE)),
     (9, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Bob</name>
                                <town>Toronto</town>
                                <street>Oak</street>
                                </doc>' PRESERVE WHITESPACE)),
     (10, XMLPARSE(DOCUMENT '<doc>
                                 <type>animal</type>
                                 <name>bird</name>
                                 </doc>' PRESERVE WHITESPACE))@
```

Procedure

You can use the following example as references when you are making your own C# CLR procedures:

- "The C# external code file" on page 109
- "Example: C# parameter style GENERAL procedure with XML features" on page 110

The C# external code file

The 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:

```
using System;
using System.IO;
using System.Data;
using IBM.Data.DB2;
using IBM.Data.DB2Types;

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. The IBM.Data.DB2Types inclusion is required if any of the procedures in the file contain parameters or variables of type XML. 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 that is 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 the procedure implementation. These names are important because the EXTERNAL clause of the **CREATE PROCEDURE** statement for each procedure must specify this information so that the database manager can locate the assembly and class of the CLR procedure.

Example: C# parameter style GENERAL procedure with XML features

The following example contains the C# code for a parameter style GENERAL procedure with XML parameter and corresponding **CREATE PROCEDURE** statement. The `xmlProc1` procedure takes two input parameters, `inNum` and `inXML`. The input parameter values are inserted into the table `xmlDataTable` and values are then retrieved using XQuery. The `xmlProc1` procedure also retries another XML value using an SQL statement. The retrieved XML values are assigned to two output parameters, `outXML1` and `outXML2`. No result sets are returned.

```
CREATE PROCEDURE  xmlProc1   ( IN inNUM  INTEGER,
                               IN inXML  XML as CLOB (1K),
                               OUT inXML  XML as CLOB (1K),
                               OUT inXML  XML as CLOB (1K)
                             )
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!xmlProc1' ;
```

```
//*************************************************************************
//  Stored Procedure: xmlProc1
//
//  Purpose:   insert XML data into XML column
//
//  Parameters:
//
//   IN:    inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
//   OUT:   outXML1 -- XML data returned - value retrieved using XQuery
//          outXML2 -- XML data returned - value retrieved using SQL
//*************************************************************************
```

```
    public static void xmlProc1 (     int      inNum, DB2Xml   inXML,
                                  out DB2Xml   outXML1, out DB2Xml   outXML2 )
    {
       // Create new command object from connection context
       DB2Parameter parm;
       DB2Command cmd;
       DB2DataReader reader = null;
       outXML1 = DB2Xml.Null;
       outXML2 = DB2Xml.Null;

       // Insert input XML parameter value into a table
       cmd = DB2Context.GetCommand();
       cmd.CommandText = "INSERT INTO "
                       + "xmlDataTable( num , xdata ) "
                       + "VALUES( ?, ? )";

       parm = cmd.Parameters.Add("@num", DB2Type.Integer );
       parm.Direction = ParameterDirection.Input;
       cmd.Parameters["@num"].Value = inNum;
       parm = cmd.Parameters.Add("@data", DB2Type.Xml);
       parm.Direction = ParameterDirection.Input;
       cmd.Parameters["@data"].Value = inXML ;
       cmd.ExecuteNonQuery();
       cmd.Close();

       // Retrieve XML value using XQuery
                     and assign value to an XML output parameter
       cmd = DB2Context.GetCommand();
```

```
        cmd.CommandText = "XQUERY for $x " +
                          "in db2-fn:xmlcolumn(\"xmlDataTable.xdata\")/doc "+
                          "where $x/make = \'Mazda\' " +
                          "return <carInfo>{$x/make}{$x/model}</carInfo>";
        reader = cmd.ExecuteReader();
        reader.CacheData= true;

        if (reader.Read())
        {  outXML1 = reader.GetDB2Xml(0);  }
        else
        {  outXML1 = DB2Xml.Null;  }

        reader.Close();
        cmd.Close();

        // Retrieve XML value using SQL
                       and assign value to an XML output parameter value
        cmd = DB2Context.GetCommand();
        cmd.CommandText = "SELECT xdata "
                           + "FROM xmlDataTable "
                           + "WHERE num = ?";

        parm = cmd.Parameters.Add("@num", DB2Type.Integer );
        parm.Direction = ParameterDirection.Input;
        cmd.Parameters["@num"].Value = inNum;
        reader = cmd.ExecuteReader();
        reader.CacheData= true;

        if (reader.Read())
        {  outXML2 = reader.GetDB2Xml(0);  }
        else
        {  outXML = DB2Xml.Null;  }

        reader.Close() ;
        cmd.Close();

        return;
    }
```

Example: XML and XQuery support in C procedure

Once the basics of procedures, the essentials of C routines, XQuery and XML are understood, you can start creating and using C procedures with XML features.

The following example demonstrates a C procedure with parameters of type XML as well as how to update and query XML data.

Prerequisites

Before working with the C procedure example you might want to read the following concept topic:

- Benefits of using routines

The following examples makes use of a table named `xmlDataTable` that is defined as follows:

```
CREATE TABLE xmlDataTable
(
   num INTEGER,
   xdata XML
)

INSERT INTO xmlDataTable VALUES
   (1, XMLPARSE(DOCUMENT '<doc>
                          <type>car</type>
                          <make>Pontiac</make>
                          <model>Sunfire</model>
                          </doc>' PRESERVE WHITESPACE)),
   (2, XMLPARSE(DOCUMENT '<doc>
                          <type>car</type>
                          <make>Mazda</make>
                          <model>Miata</model>
                          </doc>' PRESERVE WHITESPACE)),
   (3, XMLPARSE(DOCUMENT '<doc>
                          <type>person</type>
                          <name>Mary</name>
                          <town>Vancouver</town>
                          <street>Waterside</street>
                          </doc>' PRESERVE WHITESPACE)),
   (4, XMLPARSE(DOCUMENT '<doc>
                          <type>person</type>
```

```
                                <name>Mark</name>
                                <town>Edmonton</town>
                                <street>Oak</street>
                                </doc>' PRESERVE WHITESPACE)),
     (5, XMLPARSE(DOCUMENT '<doc>
                                <type>animal</type>
                                <name>dog</name>
                                </doc>' PRESERVE WHITESPACE)),
     (6, NULL),
     (7, XMLPARSE(DOCUMENT '<doc>
                                <type>car</type>
                                <make>Ford</make>
                                <model>Taurus</model>
                                </doc>' PRESERVE WHITESPACE)),
     (8, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Kim</name>
                                <town>Toronto</town>
                                <street>Elm</street>
                                </doc>' PRESERVE WHITESPACE)),
     (9, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Bob</name>
                                <town>Toronto</town>
                                <street>Oak</street>
                                </doc>' PRESERVE WHITESPACE)),
     (10, XMLPARSE(DOCUMENT '<doc>
                                 <type>animal</type>
                                 <name>bird</name>
                                 </doc>' PRESERVE WHITESPACE))
```

Procedure

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

- "The C external code file" on page 112
- "Example 1: C parameter style SQL procedure with XML features" on page 112

The C external code file

The 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 gwen`Proc.SQC` and has the following format:

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

// C procedures
               ...
```

The file inclusions are indicated at the top of the file. There are no extra include files required for XML support in embedded SQL routines.

It is important to note the name of the file and the name of the function that corresponds to the procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that the database manager can locate the library and entry point that corresponds to the C procedure.

Example 1: C parameter style SQL procedure with XML features

This example shows the following:

- CREATE PROCEDURE statement for a parameter style SQL procedure
- C code for a parameter style SQL procedure with XML parameters

This procedure receives two input parameters. The first input parameter is named `inNum` and is of type INTEGER. The second input parameters is named `inXML` and is of type XML. The values of the input parameters are used to insert a row into the table `xmlDataTable`. Then an XML value is retrieved using an SQL statement. Another XML value is retrieved using an XQuery expression. The retrieved XML values are respectively assigned to two output parameters, `out1XML` and `out2XML`. No result sets are returned.

```
CREATE PROCEDURE  xmlProc1   ( IN inNUM  INTEGER,
                               IN inXML  XML as CLOB (1K),
                               OUT inXML  XML as CLOB (1K),
                               OUT inXML  XML as CLOB (1K)
                             )
LANGUAGE C
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc!xmlProc1' ;
```

```
//*************************************************************************
//  Stored Procedure: xmlProc1
//
//  Purpose:   insert XML data into XML column
//
//  Parameters:
//
//   IN:    inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
//   OUT:   out1XML -- XML data returned - value retrieved using XQuery
//          out2XML -- XML data returned - value retrieved using SQL
//*************************************************************************
```

```
#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN testSecA1(sqlint32* inNum,
                                SQLUDF_CLOB* inXML,
                                SQLUDF_CLOB* out1XML,
                                SQLUDF_CLOB* out2XML,
                                SQLUDF_NULLIND *inNum_ind,
                                SQLUDF_NULLIND *inXML_ind,
                                SQLUDF_NULLIND *out1XML_ind,
                                SQLUDF_NULLIND *out2XML_ind,
                                SQLUDF_TRAIL_ARGS)
{
  char *str;
  FILE *file;

  EXEC SQL INCLUDE SQLCA;

  EXEC SQL BEGIN DECLARE SECTION;
       sqlint32 hvNum1;
       SQL TYPE IS XML AS CLOB(200) hvXML1;
       SQL TYPE IS XML AS CLOB(200) hvXML2;
       SQL TYPE IS XML AS CLOB(200) hvXML3;
  EXEC SQL END DECLARE SECTION;

  /* Check null indicators for input parameters */
  if ((*inNum_ind < 0) ||  (*inXML_ind < 0)) {
     strcpy(sqludf_sqlstate, "38100");
     strcpy(sqludf_msgtext, "Received null input");
     return 0;
  }

  /* Copy input parameters to host variables */
  hvNum1 = *inNum;
  hvXML1.length = inXML->length;
  strncpy(hvXML1.data, inXML->data, inXML->length);

  /* Execute SQL statement */
  EXEC SQL
     INSERT INTO xmlDataTable (num, xdata) VALUES (:hvNum1, :hvXML1);

  /* Execute SQL statement */
```

```
  EXEC SQL
     SELECT xdata INTO :hvXML2
        FROM xmlDataTable
           WHERE num = :hvNum1;

  sprintf(stmt5, "SELECT XMLQUERY('for $x in $xmldata/doc
                                   return <carInfo>{$x/model}</carInfo>'
                                   passing by ref xmlDataTable.xdata
                                       as \"xmldata\" returning sequence)
                    FROM xmlDataTable WHERE num = ?");

  EXEC SQL PREPARE selstmt5 FROM :stmt5 ;
  EXEC SQL DECLARE c5 CURSOR FOR selstmt5;
  EXEC SQL OPEN c5 using :hvNum1;
  EXEC SQL FETCH c5 INTO :hvXML3;

  exit:

  /* Set output return code */
  *outReturnCode = sqlca.sqlcode;
  *outReturnCode_ind = 0;

  return 0;
}
```

Examples of C# .NET CLR functions

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.

Before you begin

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

- ".NET common language runtime (CLR) routines" on page 75
- "Creating .NET CLR routines from the Db2 command window" on page 83
- "External scalar functions" on page 56
- Building common language runtime (CLR) .NET routines

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

About this task

For examples of CLR procedures in C#:

- "Examples of C# .NET CLR procedures" on page 91

Procedure

- Use the following examples as references when making your own C# CLR UDFs:
  - "The C# external code file" on page 114
  - "Example 1: C# parameter style SQL table function" on page 115
  - "Example 2: C# parameter style SQL scalar function" on page 116

Example

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:

```
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 `emp0ps` 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 file system. 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.

```
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
THREADSAFE
SCRATCHPAD 10
FINAL CALL
EXECUTION CONTROL SAFE
DISALLOW PARALLEL
NO DBINFO
```

```
// 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;
     }
}
```

```
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.

```
CREATE FUNCTION countUp(INTEGER)
RETURNS INTEGER
LANGUAGE CLR
PARAMETER STYLE SQL
SCRATCHPAD 10
FINAL CALL
NO SQL
FENCED
THREADSAFE
NOT DETERMINISTIC
EXECUTION CONTROL SAFE
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp' ;
```

```
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;
        }
    }
}
```

C and C++ routines

C and C++ routines are external routines that are created by executing a CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statement that references a library built from C or C++ source code as its external code body.

C and C++ routines can optionally execute SQL statements by including embedded SQL statements.

The following terms are important in the context of C and C++ routines:

CREATE statement
The SQL language CREATE statement used to create the routine in the database.

Routine-body source code
The source code file containing the C or C++ routine implementation that corresponds to the CREATE statement EXTERNAL clause specification.

Precompiler
The database utility that pre-parses the routine source code implementation to validate SQL statements contained in the code and generates a package.

Compiler
The programming language specific software required to compile and link the source code implementation.

Package
The file containing the runtime access path information that the database manager uses at routine runtime to execute the SQL statements contained in the routine code implementation.

Routine library
A file that contains the compiled form of the routine source code. In Windows this is sometimes called a DLL, because these files have .dll file extensions.

Before developing a C or C++ routine, it is important to both understand the basics of routines and the unique features and characteristics specific to C and C++ routines. An understanding of the Embedded SQL API and the basics of embedded SQL application development is also important. To learn more about these subjects, refer to the following topics:

- External routines
- Embedded SQL
- Include files for C and C++ routines
- Parameters in C and C++ routines
- Restrictions on C and C++ routines

Developing a C or C++ routines involves following a series of step by step instructions and looking at C or C++ routine examples. Refer to:

- Creating C and C++ routines
- Examples of C procedures
- Examples of C user-defined functions

Support for external routine development in C
To develop external routines in C, you must use supported compilers and development software.

The supported compilers and development software for the database application development in C can all be used for external routine development in C.

Support for external routine development in C++
To develop external routines in C++, you must use supported compilers and development software.

You can use the supported C++ compilers and C++ development software for external routine development in C++.

Tools for developing C and C++ routines
The supported development tools for C and C++ routines are the same as the tools that are supported for developing embedded SQL C and C++ applications.

The database product does not include graphical user interface tools for developing, debugging, or deploying embedded SQL applications or routines. However, you can use the following command-line interfaces for developing, debugging, and deploying embedded SQL applications and routines:

- Db2 Command Line Processor
- Db2 Command Window

These interfaces support the execution of the SQL statements that are required to create routines in a database. The PREPARE statement and the **BIND** command that are required to build C and C++ routines with embedded SQL can also be issued from these interfaces.

Designing C and C++ routines
Designing C and C++ routines is a task that should precede creating C and C++ routines. Designing C and C++ routines is generally related to both designing external routines implemented in other programming languages and designing embedded SQL applications.

Before you begin

- General knowledge of external routines
- C or C++ programming experience
- Optional: Knowledge of and experience with embedded SQL or CLI application development (if the routine will execute SQL statements)

The following topics can provide you with some of the required prerequisite information.

For more information on the features and uses of external routines:

- Refer to the topic “External routine implementation” on page 18.

For more information on the characteristics of the embedded SQL API:

- Refer to the topic, "Introduction to embedded SQL" in *Developing Embedded SQL Applications*

About this task
With the prerequisite knowledge, designing embedded SQL routines consists mainly of learning about the unique features and characteristics of C and C++ routines:

Procedure

- “Include file required for C and C++ routine development (sqludf.h)” on page 120
- “Parameters in C and C++ routines” on page 121

- “Parameter style SQL C and C++ procedures” on page 122
- “Parameter style SQL C and C++ functions” on page 125
- “SQL data type handling in C and C++ routines” on page 135
- “Graphic host variables in C and C++ routines” on page 153
- “Returning result sets from C and C++ procedures” on page 155
- “C++ type decoration” on page 153
- “Restrictions on external routines” on page 67

Results
After having learned about the C and C++ characteristics, you might want to refer to:

- “Creating C and C++ routines” on page 155

Include file required for C and C++ routine development (sqludf.h)
The `sqludf.h` include file contains structures, definitions, and values that are required when coding routine implementations.

Although the 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 `include` directory of the database installation that contains the `sqludf.h` file.

Use of objects in this file is recommended to ensure that the correct C data type for your specific operating system and operating system bit-width are used.

The `sqludf.h` file contains structure definitions and descriptions of the structure definitions. The following is a brief summary of its content:

- Macro definitions for SQL data types that are supported as parameters to external routines that do not require representation as a C or C++ structure. In the file, the definitions have name formats like: SQLUDF_*x* and SQLUDF_*x*_FBD where *x* is an SQL data type name and FBD represents FOR BIT DATA for those data types that are stored in binary form.

  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.
- C structure definitions required to represent the following SQL data types and special parameters:
  - VARBINARY
  - VARCHAR FOR BIT DATA data type
  - LONG VARCHAR data type
  - LONG VARCHAR FOR BIT DATA data type
  - LONG VARGRAPHIC data type
  - BLOB data type
  - CLOB data type
  - DBCLOB data type
  - scratchpad structure
  - dbinfo structure

  Each of these is represented by a structure with more than one field value rather than by a simple C data type.

  The scratchpad structure defines a buffer, that is passed to a user-defined function for use during the function invocation. Unlike a variable, however the data stored in a scratchpad is persistent between multiple user-defined function calls within a single invocation. This can be useful both for functions that return aggregated values and for functions that require initial setup logic.

The dbinfo structure is a structure that contains database and routine information that can be passed to and from a routine implementation as an extra argument if and only if the DBINFO clause is included in the CREATE statement for the routine.

- Definition of C language types for the scratchpad and call-type arguments. An enum type definition is specified for the call-type argument.

  External user-defined functions are invoked multiple times for a set of values. Call-types are used to identify individual external function invocations. Each invocation is identified with a call-type value that can be referenced within the function logic. For example there are special call-types for the first invocation of a function, for data fetching calls, and for the final invocation. Call-types are useful, because specific logic can be associated with a particular call-type. Examples of call-types include: FIRST call, FETCH call FINAL call.

- Macros for defining the standard trailing arguments required in user-defined function (UDF) prototypes. The trailing arguments include 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. There are various macro definitions provided that differ in their inclusion or exclusion of the scratchpad and call-type arguments. These corresponds to the presence or absence of the use of the SCRATCHPAD clause and FINAL CALL clause in the function definition.

  In general when defining a user-defined function, it is recommended to use the macro SQLUDF_TRAIL_ARGS to simplify the function prototype as shown in the following example:

```
void SQL_API_FN ScalarUDF(SQLUDF_CHAR *inJob,
                          SQLUDF_DOUBLE *inSalary,
                          SQLUDF_DOUBLE *outNewSalary,
                          SQLUDF_SMALLINT *jobNullInd,
                          SQLUDF_SMALLINT *salaryNullInd,
                          SQLUDF_SMALLINT *newSalaryNullInd,
                          SQLUDF_TRAIL_ARGS)
```

- Macro definitions that can be used to test whether or not SQL arguments have null values.

To see how the various definitions, macros, and structures that are defined in the file `sqludf.h` are used, see the C and C++ sample applications and routines.

Parameters in C and C++ routines

Parameter declaration in C and C++ routines must conform to the requirements of one of the supported parameter styles and the program type.

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 including:

- "Parameter styles supported for C and C++ routines" on page 122
- "Parameter null indicators in C and C++ routines" on page 122
- "Parameter style SQL C and C++ procedures" on page 122
- "Parameter style SQL C and C++ functions" on page 125
- "Passing parameters by value or by reference in C and C++ routines" on page 126
- "Parameters are not required for C and C++ procedure result sets" on page 127
- "The dbinfo structure as C or C++ routine parameter" on page 127
- "Scratchpad as C or C++ function parameter" on page 129
- "Program type MAIN support for C and C++ procedures" on page 130

It is very important that you implement the parameter interface to C and C++ routines correctly. This can be easily done with just a bit of care taken to ensure that the correct parameter style and data types are chosen and implemented according to the specification.

Parameter styles supported for C and C++ routines
For C and C++ routines, the SQL parameter style is supported for procedures and functions, and the GENERAL and GENERAL WITH NULLS parameter styles are supported for only procedures.

It is strongly recommended that the parameter style SQL be used for all C and C++ routines. This parameter style supports NULL values, provides a standard interface for reporting errors, as well as supporting scratchpads and call types.

To specify the parameter style to be used for a routine, you must specify the PARAMETER STYLE clause in the CREATE statement for the routine at routine creation time.

The parameter style must be accurately reflected in the implementation of the C or C++ routine code.

For more information about these parameter styles refer to: "Syntax for passing parameters to C and C++ routines".

Parameter null indicators in C and C++ routines
If the parameter style chosen for a C or C++ routine (procedure or function) requires that a null indicator parameter be specified for each of the SQL parameters, as is required by parameter style SQL and GENERAL, the null indicators are to be passed as parameters of data type SQLUDF_NULLIND*. For parameter style GENERAL WITH NULLS, they must be passed as an array of type SQLUDF_NULLIND.

This data type is defined in embedded SQL application and routine include file: `sqludf.h`.

Null-indicator parameters indicate whether the corresponding parameter value is equivalent to NULL in SQL or if it has a literal value. If the null indicator value for a parameter is 0, this indicates that the parameter value is not null. If the null-indicator value for a parameter is -1, the parameter is to be considered to have a value equivalent to the SQL value NULL.

When null indicators are used it is important to include code within your routine that:

- Checks null-indicator values for input parameters before using them.
- Sets null indicator values for output parameters before the routine returns.

For more information about parameter SQL refer to:

- “External routine parameter styles” on page 65
- “Parameter style SQL C and C++ procedures” on page 122
- “Parameter style SQL C and C++ functions” on page 125

Parameter style SQL C and C++ procedures
C and C++ procedures can be created with the PARAMETER STYLE SQL clause in the CREATE PROCEDURE statement.

The C and C++ PARAMETER STYLE SQL signature for a procedure must be in the following 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 or C++ procedure, which can vary across supported operating systems. The use of these macros is required for C and C++ routines. The macros are declared in the embedded SQL application and routine include file `sqlsystm.h`.

function-name
Name of the C or C++ function within the code file. This value does not have to be the same as the name of the procedure that is specified within the corresponding CREATE PROCEDURE statement. This value in combination with the library name however must be specified in the EXTERNAL NAME clause to identify the correct function entry point within the library to be used. 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 must be defined as `extern "C"` in the user code. The function name must be explicitly exported.

SQL-arguments
C or C++ arguments that correspond to the set of SQL parameters that are specified in the CREATE PROCEDURE statement. IN, OUT, and INOUT mode parameters are passed using individual pointer values.

SQL-argument-inds
C or C++ null indicators that correspond to the set of SQL parameters that are specified in the CREATE PROCEDURE statement. For each IN, OUT, and INOUT mode parameter, there must be an associated null-indicator parameter. Null indicators can be passed as individual arguments of type SQLUDF_NULLIND or as part of a single array of null indicators that are defined as SQLUDF_NULLIND*.

sqlstate
Input-output parameter value that is used by the routine to signal warning or error conditions. Typically this argument is used to assign a user-defined SQLSTATE value that corresponds to an error or a warning that can be passed back to the caller. SQLSTATE values of the form `38xxx`, where `xxx` is any numeric value, are available for user-defined SQLSTATE error values. SQLSTATE values of the form `01Hxx`, where `xx` is any numeric value, are available for user-defined SQLSTATE warning values.

routine-name
Input parameter value that contains the qualified routine name. This value is generated by the database manager and passed to the routine in the form `<schema-name>.<routine-name>` where `<schema-name>` and `<routine-name>` correspond respectively to the ROUTINESCHEMA column value and ROUTINENAME column value for the routine within the SYSCAT.ROUTINES catalog view. This value can be useful if a single routine implementation is used by multiple different routine definitions. When the routine definition name is passed into the routine, logic can be conditionally executed based on which definition was used. The routine name can also be useful when you formulate diagnostic information such as error messages, or log file entries.

specific-name
Input parameter value that contains the unique routine specific name. This value is generated by the database manager and passed to the routine. This value corresponds to the SPECIFICNAME column value for the routine in the SYSCAT.ROUTINES view. It can be useful in the same way as the routine-name.

diagnostic-message
Output parameter value that is optionally used by the routine to return message text to the caller application or routine. This parameter is intended to be used as a complement to the SQLSTATE argument. It can be used to assign a user-defined error-message to accompany a user-defined SQLSTATE value, which can provide more detailed diagnostic error or warning information to the caller of the routine.

Remember: You can use the macro definition SQLUDF_TRAIL_ARGS, which is defined in the `sqludf.h` file, in place of using individual arguments for implementing the non-SQL data type arguments to simplify the writing of C and C++ procedure signature.

The following sample C or C++ procedure accepts a single input parameter, and returns a single output parameter and a result set:

```
/*****************************************************************
 Routine:   cstp

  Purpose:  Returns an output parameter value based on an input
            parameter value

            Shows how to:
             - define a procedure using PARAMETER STYLE SQL
             - define NULL indicators for the parameter
             - execute an SQL statement
             - how to set a NULL indicator when parameter is
               not null

  Parameters:
```

```
  IN:       inParm
  OUT:      outParm

            When PARAMETER STYLE SQL is defined for the routine
            (see routine registration script spcreate.db2), in
            addition to the parameters passed during invocation,
            the following arguments are passed to the routine
            in the following order:

             - one null indicator for each IN/INOUT/OUT parameter
               ordered to match order of parameter declarations
             - SQLSTATE to be returned to database (output)
             - qualified name of the routine (input)
             - specific name of the routine (input)
             - SQL diagnostic string to return an optional
               error message text to database (output)

            See the actual parameter declarations below to see
            the recommended datatypes and sizes for them.

            CODE TIP:
            --------
            Instead of coding the 'extra' parameters:
              sqlstate, qualified name of the routine,
              specific name of the routine, diagnostic message,
            a macro SQLUDF_TRAIL_ARGS can be used instead.
            This macro is defined in database include file sqludf.h

            TIP EXAMPLE:
            ------------
            The following is equivalent to the actual prototype
            used that makes use of macro definitions included in
            sqludf.h.  The form actually implemented is simpler
            and removes datatype concerns.

   extern "C"  SQL_API_RC SQL_API_FN OutLanguage(
                                                 sqlint16 *inParm,
                                                 double *outParm,
                                                 sqlint16 *inParmNullInd,
                                                 sqlint16 *outParmNullInd,
                                                 char sqlst[6],
                                                 char qualName[28],
                                                 char specName[19],
                                                 char diagMsg[71])
                                                )

*****************************************************************/

extern "C" SQL_API_RC SQL_API_FN cstp ( sqlint16 *inParm,
                                        double *outParm,
                                        SQLUDF_NULLIND *inParmNullInd,
                                        SQLUDF_NULLIND *outParmNullInd,
                                        SQLUDF_TRAIL_ARGS )
{
  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;
  *outParmNullInd = 0;

  EXEC SQL OPEN cur1;

  return (0);
}
```

The corresponding CREATE PROCEDURE statement for this procedure 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 create procedure statement `cproc` assumes that the C or C++ procedure implementation is in a library file that is named `c_rtns` and a function named `cstp`.

Parameter style SQL C and C++ functions
C and C++ user-defined functions should be created using the PARAMETER STYLE SQL clause in the CREATE FUNCTION statement. The parameter passing conventions of this parameter style should be implemented in the corresponding source code implementation.

The C and C++ PARAMETER STYLE SQL signature implementation required for user-defined functions 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 or C++ user-defined function, which can vary across supported operating systems. The use of these macros is required for C and C++ routines. The macros are declared in embedded SQL application and routine include file `sqlsystm.h`.

function-name
Name of the C or C++ function within the code file. This value does not have to be the same as the name of the function specified within the corresponding CREATE FUNCTION statement. This value in combination with the library name however must be specified in the EXTERNAL NAME clause to identify the correct function entry point within the library to be used. 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 function declaration within the source code file should be prefixed with `extern "C"` as shown in the following example: extern "C" SQL_API_RC SQL_API_FN OutLanguage(char *, sqlint16 *, char *, char *, char *, char *);

SQL-arguments
C or C++ arguments that correspond to the set of SQL parameters specified in the CREATE FUNCTION statement.

SQL-argument-inds
For each SQL-argument a null indicator parameter is required to specify whether the parameter value is intended to be interpreted within the routine implementation as a NULL value in SQL. Null indicators must be specified with data type SQLUDF_NULLIND. This data type is defined in embedded SQL routine include file `sqludf.h`.

SQLUDF_TRAIL_ARGS
A macro defined in embedded SQL routine include file `sqludf.h` that once expanded defines the additional trailing arguments required for a complete parameter style SQL signature. There are two macros that can be used: SQLUDF_TRAIL_ARGS and SQLUDF_TRAIL_ARGS_ALL. SQLUDF_TRAIL_ARGS when expanded, as defined in `sqludf.h`, is equivalent to the addition of the following routine arguments:

```
SQLUDF_CHAR *sqlState,
SQLUDF_CHAR qualName,
SQLUDF_CHAR specName,
SQLUDF_CHAR *sqlMessageText,
```

In general these arguments are not required or generally used as part of user-defined function logic. They represent the output SQLSTATE value to be passed back to the function invoker, the input fully qualified function name, input function specific name, and output message text to be returned with

the SQLSTATE. SQLUDF_TRAIL_ARGS_ALL when expanded, as defined in `sqludf.h`, is equivalent to the addition of the following routine arguments:

```
SQLUDF_CHAR  qualName,
SQLUDF_CHAR  specName,
SQLUDF_CHAR  sqlMessageText,
SQLUDF_SCRAT *scratchpad
SQLUDF_CALLT *callType
```

If the UDF CREATE statement includes the SCRATCHPAD clause or the FINAL CALL clause, then the macro SQLUDF_TRAIL_ARGS_ALL must be used. In addition to arguments provided with SQLUDF_TRAIL_ARGS, this macro also contains pointers to a scratchpad structure, and a call type value.

The following is an example of a simple C or C++ UDF that returns in an output parameter the value of the product of its two input parameter values:

```
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 )
{

  /* Check that input parameter values are not null
     by checking the corresponding null indicator values
         0  : indicates parameter value is not NULL
         -1 : indicates parameter value is NULL

    If values are not NULL, calculate the product.
    If values are NULL, return a NULL output value. */

  if ((*in1NullInd != -1) &&
       *in2NullInd != -1))
  {
    *outProduct = (*in1) * (*in2);
    *productNullInd = 0;
  }
  else
  {
    *productNullInd = -1;
  }
  return (0);
}
```

The corresponding CREATE FUNCTION statement that can be used to create this UDF could be:

```
CREATE FUNCTION product( in1 DOUBLE, in2 DOUBLE )
  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 SQL statement assumes that the C or C++ function is in a library file in the function directory named `c_rtns`.

Passing parameters by value or by reference in C and C++ routines
For C and C++ routines, parameter values must always be passed by reference to routines using pointers.

This is required for input-only, input-output, and output parameters by reference.

Null-indicator parameters must also be passed by reference to routines using pointers.

The database manager control the allocation of memory for all parameters and maintain C or C++ references to all parameters that are passed into or out of a routine. There is no need to allocate or free memory that is associated with routine parameters and null indicators.

Parameters are not required for C and C++ procedure result sets
No parameter is required in the CREATE PROCEDURE statement signature for a procedure or in the associated procedure implementation in order to return a result set to the caller.

Result sets returned from C procedures, are returned using cursors.

For more on returning result sets from LANGUAGE C procedures, see:

- "Returning result sets from C and C++ procedures" on page 155

The dbinfo structure as C or C++ routine parameter
The dbinfo structure is a structure that contains database and routine information that can be passed to and from a routine implementation as an extra argument if and only if the DBINFO clause is included in the CREATE statement for the routine.

The dbinfo structure is supported in LANGUAGE C routines by using the sqludf_dbinfo structure. The dbinfo C structure is defined in the sqludf.h file that is located in the sqllib\include directory.

The sqludf_dbinfo structure is defined as follows:

```
SQL_STRUCTURE sqludf_dbinfo
{
   unsigned short  dbnamelen;                      /* Database name length     */
   unsigned char   dbname[SQLUDF_MAX_IDENT_LEN];   /* Database name            */
   unsigned short  authidlen;                      /* Authorization ID length  */
   unsigned char   authid[SQLUDF_MAX_IDENT_LEN];   /* Authorization ID         */
   union db_cdpg   codepg;                         /* Database code page       */
   unsigned short  tbschemalen;                    /* Table schema name length */
   unsigned char   tbschema[SQLUDF_MAX_IDENT_LEN]; /* Table schema name        */
   unsigned short  tbnamelen;                      /* Table name length        */
   unsigned char   tbname[SQLUDF_MAX_IDENT_LEN];   /* Table name               */
   unsigned short  colnamelen;                     /* Column name length       */
   unsigned char   colname[SQLUDF_MAX_IDENT_LEN];  /* Column name              */
   unsigned char   ver_rel[SQLUDF_SH_IDENT_LEN];   /* Database version/release */
   unsigned char   resd0[2];                       /* Alignment                */
   sqluint32       platform;                       /* Platform                 */
   unsigned short  numtfcol;                       /* # of entries in TF column*/
                                                   /* List array               */
   unsigned char   resd1[2];                       /* Reserved                 */
   sqluint32       procid;                         /* Current procedure ID     */
   unsigned char   resd2[32];                      /* Reserved                 */
   unsigned short  *tfcolumn;                      /* Tfcolumn to be allocated */
                                                   /* dynamically if a table   */
                                                   /* function is defined;     */
                                                   /* else a NULL pointer      */
   char            *appl_id;                       /* Application identifier   */
   sqluint32       dbpartitionnum;                 /* Database partition number*/
                                                   /* where routine executed   */


   sqluint32       numdbpartitions;                /* number of entries in     */
                                                   /* dbpartitions array       */
   sqluint32       *dbpartitions;                  /* allocated dynamically if */
                                                   /* routine is processed in  */
                                                   /* parallel. Otherwise, this*/
                                                   /* will be a null pointer.  */

   unsigned char   resd3[16];                      /* Reserved                 */
};
```

Although not all of the fields in the dbinfo structure might be useful within a routine, several of the values in the structure fields might be useful when formulating diagnostic error message information. For example, if an error occurs within a routine, it might be useful to return the database name, database name length, the database code page, the current authorization ID, and the length of the current authorization ID.

To reference the sqludf_dbinfo structure in a LANGUAGE C routine implementation:

- Add the DBINFO clause to the CREATE statement that defines the routine.
- Include the sqludf.h header file at the top of the file containing the routine implementation.
- Add a parameter of type sqludf_dbinfo to the routine signature in the position specified by the parameter style used.

Example of a C procedure using the dbinfo structure

The following example of a C procedure with PARAMETER STYLE GENERAL demonstrates the use of the dbinfo structure.

Here is the CREATE PROCEDURE statement for the procedure. The procedure implementation is located in a library file named spserver that contains a C function named DbinfoExample, as specified by the EXTERNAL NAME clause:

```
  CREATE PROCEDURE DBINFO_EXAMPLE (IN job CHAR(8),
                                   OUT salary DOUBLE,
                                   OUT dbname CHAR(128),
                                   OUT dbversion CHAR(8),
                                   OUT errorcode INTEGER)
  DYNAMIC RESULT SETS 0
  LANGUAGE C
  PARAMETER STYLE GENERAL
  DBINFO
  FENCED NOT THREADSAFE
  READS SQL DATA
  PROGRAM TYPE SUB
  EXTERNAL NAME 'spserver!DbinfoExample'@
```

Here is the C procedure implementation that corresponds to the procedure definition:

```
/*****************************************************************
 Routine:    DbinfoExample

   IN:       inJob         - a job type, used in a SELECT predicate
   OUT:      salary        - average salary of employees with job injob
             dbname        - database name retrieved from DBINFO
             dbversion     - database version retrieved from DBINFO
             outSqlError   - sqlcode of error raised (if any)
             sqludf_dbinfo - pointer to DBINFO structure

  Purpose:  This routine takes in a job type and returns the
            average salary of all employees with that job, as
            well as information about the database (name,
            version of database).  The database information
            is retrieved from the dbinfo object.

  Shows how to:
              - define IN/OUT parameters in PARAMETER STYLE GENERAL
              - declare a parameter pointer to the dbinfo structure
              - retrieve values from the dbinfo structure
*****************************************************************/
SQL_API_RC SQL_API_FN DbinfoExample(char inJob[9],
                                    double *salary,
                                    char dbname[129],
                                    char dbversion[9],
                                    sqlint32 *outSqlError,
                                    struct sqludf_dbinfo * dbinfo
                                    )
{
  /* Declare a local SQLCA */
  struct sqlca sqlca;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  /* SQL host variable declaration section */
  /* Each host variable names must be unique within a code
     file, or the precompiler raises SQL0307 error */
  EXEC SQL BEGIN DECLARE SECTION;
  char dbinfo_injob[9];
  double dbinfo_outsalary;
  sqlint16 dbinfo_outsalaryind;
  EXEC SQL END DECLARE SECTION;

  /* Initialize output parameters - se strings to NULL */
  memset(dbname, '\0', 129);
  memset(dbversion, '\0', 9);
  *outSqlError = 0;

  /* Copy input parameter into local host variable */
  strcpy(dbinfo_injob, inJob);

  EXEC SQL SELECT AVG(salary) INTO:dbinfo_outsalary
             FROM employee
```

```
               WHERE job =:dbinfo_injob;

  *salary = dbinfo_outsalary;

  /* Copy values from the DBINFO structure into the output parameters
     You must explicitly null-terminate the strings.
     Information such as the database name, and the version of the
     database product can be found in the DBINFO structure as well as
     other information fields. */

  strncpy(dbname, (char *)(dbinfo->dbname), dbinfo->dbnamelen);
  dbname[dbinfo->dbnamelen] = '\0';
  strncpy(dbversion, (char *)(dbinfo->ver_rel), 8);
  dbversion[8] = '\0';

  return 0;

  /* Copy SQLCODE to OUT parameter if SQL error occurs */

  return_error:
  {
    *outSqlError = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return 0;
  }
} /* DbinfoExample function */
```

Scratchpad as C or C++ function parameter
The scratchpad structure, used for storing UDF values between invocations for each UDF input value, is supported in C and C++ routines through the use of the sqludf_scrat structure.

The `sqludf_scrat` C structure is defined in the include file `sqludf.h`.

To reference the sqludf_scrat structure, include the `sqludf.h` header file at the top of the file containing the C or C++ function implementation, and use the SQLUDF_TRAIL_ARGS_ALL macro within the signature of the routine implementation.

The following example demonstrates a C scalar function implementation that includes a parameter of type SQLUDF_TRAIL_ARGS_ALL:

```
  #ifdef __cplusplus
  extern "C"
  #endif
  void SQL_API_FN ScratchpadScUDF(SQLUDF_INTEGER *outCounter,
                                  SQLUDF_SMALLINT *counterNullInd,
                                  SQLUDF_TRAIL_ARGS_ALL)
  {
    struct scalar_scratchpad_data *pScratData;

    /* SQLUDF_CALLT and SQLUDF_SCRAT are */
    /* parts of SQLUDF_TRAIL_ARGS_ALL */

    pScratData = (struct scalar_scratchpad_data *)SQLUDF_SCRAT->data;
    switch (SQLUDF_CALLT)
    {
      case SQLUDF_FIRST_CALL:
        pScratData->counter = 1;
        break;
      case SQLUDF_NORMAL_CALL:
        pScratData->counter = pScratData->counter + 1;
        break;
      case SQLUDF_FINAL_CALL:
        break;
    }

    *outCounter = pScratData->counter;
    *counterNullInd = 0;
  } /* ScratchpadScUDF */
```

The SQLUDF_TRAIL_ARGS_ALL macro expands to define other parameter values including one called SQLUDF_SCRAT that defines a buffer parameter to be used as a scratchpad. When the scalar function is invoked for a set of values, for each time the scalar function is invoked, the buffer is passed as a parameter to the function. The buffer can be used to be accessed

The SQLUDF_TRAIL_ARGS_ALL macro value also defines another parameter SQLUDF_CALLT. This parameter is used to indicate a call type value. Call type values can be used to identify if a function is being invoked for the first time for a set of values, the last time, or at a time in the middle of the processing.

Program type MAIN support for C and C++ procedures
Although the default PROGRAM TYPE clause value SUB is generally recommended for C procedures, the PROGRAM TYPE clause value MAIN is supported in CREATE PROCEDURE statements where the LANGUAGE clause value is C.

The PROGRAM TYPE clause value MAIN is required for routines with greater than ninety parameters.

When a PROGRAM TYPE MAIN clause is specified, procedures must be implemented using a signature that is consistent with the default style for a main routine in a C source code file. This does not mean that the routine must be implemented by a function named `main`, but rather that the parameters be passed in the format generally associated with a default type main routine application implementation that uses typical C programming `argc` and `argv` arguments.

Here is an example of a C or C++ routine signature that adheres to the PGRAM TYPE MAIN specification:

```
SQL_API_RC SQL_API_FN functionName(int argc, char **argv)
{
  ...
}
```

The total number of arguments to the function is specified by the value of argc. The argument values are passed as array elements within the argv array. The number and order of the arguments depends on the PARAMETER STYLE clause value specified in the CREATE PROCEDURE statement.

As an example, consider the following CREATE PROCEDURE statement for a C procedure specified to have a PROGRAM TYPE MAIN style and the recommended PARAMETER STYLE SQL:

```
CREATE PROCEDURE MAIN_EXAMPLE (
  IN job CHAR(8),
  OUT salary DOUBLE)
SPECIFIC CPP_MAIN_EXAMPLE
DYNAMIC RESULT SETS 0
NOT DETERMINISTIC
LANGUAGE C
PARAMETER STYLE SQL
NO DBINFO
FENCED NOT THREADSAFE
READS SQL DATA
PROGRAM TYPE MAIN
EXTERNAL NAME 'spserver!MainExample'@
```

The routine signature implementation that corresponds to this CREATE PROCEDURE statement follows:

```
//*************************************************
//  Stored Procedure: MainExample
//
//  SQL parameters:
//     IN:      argv[1] - job    (char[8])
//     OUT:     argv[2] - salary (double)
//*************************************************
SQL_API_RC SQL_API_FN MainExample(int argc, char **argv)
{
  ...
}
```

Because PARAMETER STYLE SQL is used, in addition to the SQL parameter values passed at procedure invocation time, the additional parameters required for that style are also passed to the routine.

Parameter values can be accessed by referencing the argv array element of interest within the source code. For the example given previously, the argc and the argv array elements contain the following values:

```
argc   : Number of argv array elements
argv[0]: The function name
argv[1]: Value of parameter job (char[8], input)
```

```
    argv[2]: Value of parameter salary (double, output)
    argv[3]: null indicator for parameter job
    argv[4]: null indicator for parameter salary
    argv[5]: sqlstate (char[6], output)
    argv[6]: qualName (char[28], output)
    argv[7]: specName (char[19], output)
    argv[8]: diagMsg (char[71], output)
```

Supported SQL data types in C and C++ routines
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 9. SQL Data Types Mapped to C/C++ Declarations

SQL Column Type	C/C++ Data Type	SQL Column Type Description
SMALLINT	sqlint16 SQLUDF_SMALLINT	16-bit signed integer
INTEGER	sqlint32 SQLUDF_INTEGER	32-bit signed integer
BIGINT	sqlint64 SQLUDF_BIGINT	64-bit signed integer
REAL FLOAT(*n*) where 1<=*n*<=24	float SQLUDF_REAL	Single-precision floating point
DOUBLE FLOAT FLOAT(*n*) where 25<=*n*<=53	double SQLUDF_DOUBLE	Double-precision floating point
DECIMAL(*p*, *s*)	Not supported.	To pass a decimal value, define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type.
CHAR(*n*)	char[*n*+1] where *n* is large enough to hold the data 1<=*n*<=255 SQLUDF_CHAR	Fixed-length, null-terminated character string
CHAR(*n*) FOR BIT DATA	char[*n*] where *n* is large enough to hold the data 1<=*n*<=255 SQLUDF_CHAR	Fixed-length, not 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

Table 9. SQL Data Types Mapped to C/C++ Declarations (continued)

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

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

BINARY	char[*n*] where *n* is large enough to hold the data 1<=*n*<=255 SQLUDF_BINARY	Fixed-length, not null-terminated binary string

VARBINARY	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
	
	SQLUDF_VARBINARY
		Not null-terminated varying length binary string

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

Table 9. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
DATE	char[11] SQLUDF_DATE	Null-terminated character string of the following format: `yyyy-mm-dd`
TIME	char[9] SQLUDF_TIME	Null-terminated character string of the following format: `hh.mm.ss`

TIMESTAMP	char[20] - char[33]
	SQLUDF_STAMP
		Null-terminated character string of the following format:
		`yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn`
		The character string can be from 19 - 32 bytes in length depending on the number of fractional seconds specified. The fractional seconds of the TIMESTAMP data type can be optionally specified with 0-12 digits of timestamp precision.
		For example:
		`(VALUES(CURRENT TIMESTAMP(0))`
		`1`
		`-------------------`
		`2008-07-09-14.48.36`
		`1 record(s) selected.`
		`LENGTH (VALUES(CURRENT TIMESTAMP(0))`
		`1`
		`-------------------`
		`19`
		`1 record(s) selected.`
		`(VALUES(CURRENT TIMESTAMP(12))`
		`1`
		`-------------------`
		`2008-07-09-14.48.36.123456789012`
		`1 record(s) selected.`
		`LENGTH (VALUES(CURRENT TIMESTAMP(0))`
		`1`
		`-------------------`
		`32`
		When a timestamp value is assigned to a timestamp variable with a different number of fractional seconds, the value is either truncated or padded with 0's to match the format of the timestamp variable.

LOB LOCATOR	sqluint32 SQLUDF_LOCATOR	32-bit signed integer

Table 9. SQL Data Types Mapped to C/C++ Declarations (continued)

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

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

Note: XML data types can only be implemented as CLOB data types in external routines implemented in C or C++.

Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option:

- GRAPHIC(*n*)
- VARGRAPHIC(*n*)
- LONG VARGRAPHIC
- DBCLOB(*n*)

SQL data type handling in C and 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 the database 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 SMALLINT value that is returned from the routine body is cast to an INTEGER data type before it is passed 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 the database 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 */
```

The database 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, because the database 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 */
```

Input routine parameters of data type CHAR are always automatically null terminated. For a CHAR(n) input parameter, where *n* is the length of the CHAR data type, *n* bytes of data are moved to the buffer in the routine implementation and the character in the *n* + 1 position is set to the ASCII null terminator character (X'00').

Output parameters of procedures and return values of functions of data type CHAR must be explicitly null terminated by the routine. For a return value of a UDF specified by the RETURNS clause, such as RETURNS CHAR(n), or a procedure output parameter specified as CHAR(n), where n is the length of the CHAR value, a null terminator character must exist within the first *n*+1 bytes of the buffer. If a null terminator is found within the first *n*+1 bytes of the buffer, the remaining bytes, up to byte *n*, are set to ASCII blank characters X'20'). If no null terminator is found, an SQL error (SQLSTATE 39501) results.

For input and output parameters of procedures or function return values of data type CHAR that also specify the FOR BIT DATA clause, which indicates that the data is to be manipulated in its binary form, null terminators are not used to indicate the end of the parameter value. For either a RETURNS CHAR*n*) FOR BIT DATA function return value or a CHAR(*n*) FOR BIT DATA output parameter, the first *n* bytes of the buffer are copied over regardless of any occurrences of string null terminators within the first *n* bytes. Null terminator characters identified within the buffer are ignored as null terminators and instead are simply 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 the database 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, the database places 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 the database determines 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 */
```

  The characters for the DATE, TIME, and TIMESTAMP return values must be in the defined form to avoid misinterpretation of the value by the database. For example, 2001-04-03 can be interpreted as April 3 even if March 4 is intended.

- 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 TIME          */
     char     result[9];     /* also acceptable */
```

- TIMESTAMP

  Valid. Represent in C as SQLUDF_STAMP or as CHAR(19) - CHAR(32), that is, as `char[20] to char[33]`. The timestamp value has the following format:

```
yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn
```

  where:

  yyyy
  Represents the year.

  mm
  Represents the month.

  dd
  Represents the day.

  hh
  Represents the hour.

  mm
  Represents the minutes.

  ss
  Represents the seconds.

nnnnnnnnnnnn
Represents the fractional seconds. The fractional seconds of the TIMESTAMP data type can be optionally specified with 0-12 digits of timestamp precision.

When a timestamp value is assigned to a timestamp variable with a different number of fractional seconds, the value is either truncated or padded with 0's to match the format of the timestamp variable.

The character string can be from 19 - 32 bytes in length depending on the number of fractional seconds specified. The fractional seconds of the TIMESTAMP data type can be optionally specified with 0-12 digits of timestamp precision.

For example:

```
(VALUES(CURRENT TIMESTAMP(0))
1
-------------------
2008-07-09-14.48.36

  1 record(s) selected.

LENGTH (VALUES(CURRENT TIMESTAMP(0))
1
-------------------
19

  1 record(s) selected.
(VALUES(CURRENT TIMESTAMP(12))
1
-------------------
2008-07-09-14.48.36.123456789012

  1 record(s) selected.

LENGTH (VALUES(CURRENT TIMESTAMP(0))
1
-------------------
32
```

The following are variable declarations that can hold a TIMESTAMP(12) value:

```
     char     arg1[33];      /* 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, the database moves *n* double-byte characters to the buffer and sets the following two bytes to null. Data passed from the database 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, the database looks for an embedded GRAPHIC null CHAR, and if it finds it, pads the value out to *n* with GRAPHIC blank characters.

When you define graphic routine parameters, consider using VARGRAPHIC rather than GRAPHIC as the database do not promote VARGRAPHIC arguments to GRAPHIC. For example, suppose that 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, the database places 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 the database 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 the database determines 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 the database 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   */
```

- BINARY

  Valid. Represent in C as SQLUDF_BINARY.

  Example:

```
SQLUDF_BINARY     arg1[14];      /* example for BINARY(14)   */
SQLUDF_BINARY    *arg1;          /* also acceptable          */
```

  Input routine parameters of data type BINARY are always automatically null terminated. For a BINARY(n) input parameter where n is the length of the BINARY data type, n bytes of the data are

moved to the buffer in the routine implementation and the byte in the n+1 position is set to the ASCII null terminator value (x'00').

Output parameters of procedures and return values of functions of data type BINARY must be explicitly null terminated by the routine. For a return value of a UDF specified by the RETURNS clause, such as RETURNS BINARY(n), or a procedure output parameter that is specified as BINARY(n), where n is the length of the BINARY value, a null terminator character must exist within the first *n*+1 bytes of the buffer. If a null terminator is found within the first *n*+1 bytes of the buffer, the remaining bytes, up to byte *n*, are set to ASCII blank characters X'20').

- VARBINARY(n)

  Valid. Represent in C as SQLUDF_VARBINARY. The SQLUDF_VARBINARY type is defined as the sqludf_vc_fbd structure type.

```
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 VARBINARY */
```

- 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 the database 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) */
```

- XML

  Valid. Represent in C as SQLUDF_XML or in the way as a CLOB data type is represented; that is with 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 XML(n) */
struct sqludf_lob *result;
```

  The assignment and access of XML parameter and variable values in C and C++ external routine code is done in the same way as for CLOB values.

- 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.

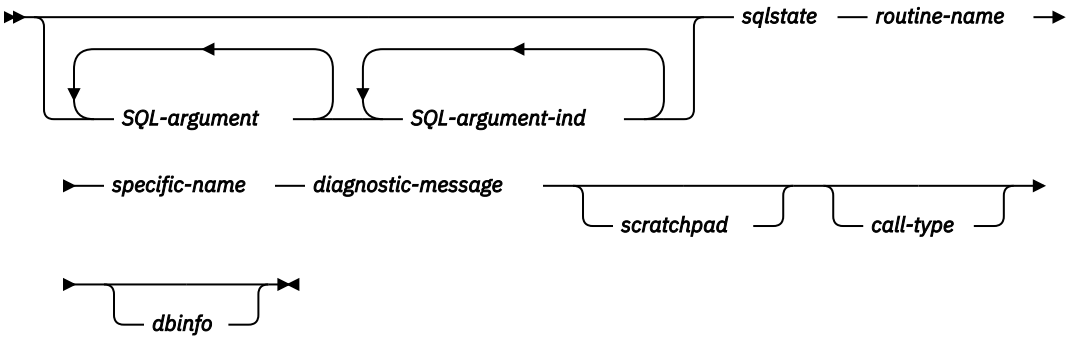
How to pass arguments to C, C++, OLE, or COBOL routines

In addition to the SQL arguments that are specified in the DML reference for a routine, the database manager pass 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.

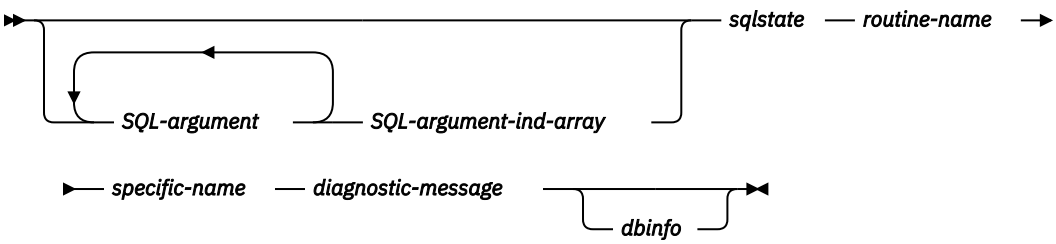
To ensure that information is exchanged correctly between invokers and the routine body, you must ensure that your routine accepts arguments in the order they are passed, according to the parameter style that is being used. The `sqludf` include file can aid you in handling and using these arguments.

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

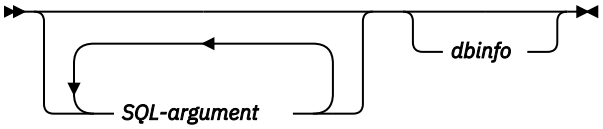
PARAMETER STYLE SQL routines



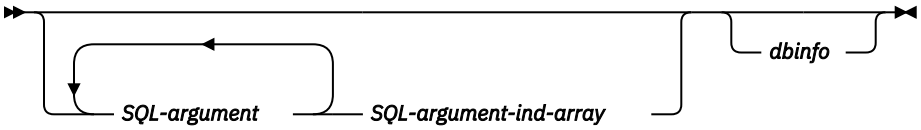
PARAMETER STYLE DB2SQL procedures



PARAMETER STYLE GENERAL procedures



PARAMETER STYLE GENERAL WITH NULLS procedures



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

The arguments for the PARAMETER STYLE SQL, PARAMETER STYLE GENERAL, and PARAMETER STYLE GENERAL WITH NULL are described as follows:

SQL-argument...

Each *SQL-argument* argument represents one input or output value that is 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 a function followed by one *SQL-argument* argument for the result of a function.
- For a table function, one argument for each input parameter to a function followed by one *SQL-argument* argument for each column in the result table of a function.
- For a method, one *SQL-argument* argument for the subject type of a method, then one argument for each input parameter to a method followed by one *SQL-argument* for the result of a method.
- For a stored procedure, one *SQL-argument* argument for each parameter to a stored procedure.

Each *SQL-argument* 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.

  The input argument is set by the database manager before the routine is called. The value of each of these arguments is taken from the expression that is 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, method, or an OUT parameter of a stored procedure.

  The output argument is set by a routine before it is returned to the database manager. The database manager allocates the buffer and passes its address to a routine. The routine puts the result value into the buffer. Enough buffer space is allocated by the database manager to contain the value that is expressed in the data type. For character types and LOBs, 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 the CAST FROM clause is not present, then the result data type is defined in the RETURNS clause.

  For table functions, the database manager defines a performance optimization where every defined column does not have to be returned to the database manager. If you write your UDF to take advantage of this feature, it returns only the columns that are required by the statement, which references the table function. For example, consider a CREATE FUNCTION statement for a table function that is defined with 100 result columns. If a statement that references the function is only interested in two columns, optimization by the database manager enables the UDF to return only those two columns for each row and not spend time on the other 98 columns. For more information, see the dbinfo argument content.

  For each value returned, the routine returns bytes that are 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.

  The INOUT argument behaves as both an IN and an OUT parameter and therefore follows both sets of rules that are shown previously. The database manager sets the argument before a stored procedure is called. The buffer for the argument is allocated by the database manager based on the maximum size of the parameter that is defined in the CREATE procedure statement. For example, an INOUT parameter of a CHAR type might 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 it is returned to the database manager.

SQL-argument-ind...
The *SQL-argument-ind* argument corresponds to each *SQL-argument* value that is passed to a routine. The *n*th *SQL-argument-ind* argument corresponds to the *n*th *SQL-argument* value and indicates whether the *SQL-argument* argument contains a NULL value.

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:

  The input argument is set by the database manager before a routine is called. It contains one of the following values:

  0
  The argument is present and it is not NULL.

  -1
  The argument is present and its value is NULL.

  If the routine is defined with the `RETURNS NULL ON NULL INPUT` option, the routine body does not need to check for a NULL value. However, if the routine is defined with the `CALLED ON NULL INPUT` option, any argument can be NULL and the routine must check the *SQL-argument-ind* value before it uses the corresponding *SQL-argument* value.

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

  The output argument is set by the routine before it is returned to the database manager. The *SQL-argument-ind* 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 the `RETURNS NULL ON NULL INPUT` option, the routine body must set the *SQL-argument-ind* value of the result. For example, a divide function might set the result to null when the denominator is zero.

  For scalar functions and methods, the database manager treats a NULL result as an arithmetic error if the following conditions are true:

  – The database configuration parameter **dft_sqlmathwarn** is set to YES.
  – One of the input arguments is a null because of an arithmetic error.

  The database manager treats NULL results as an arithmetic error if you define the function with the `RETURNS NULL ON NULL INPUT` option.

  For table functions that use the result column list to optimize performance, only the indicators corresponding to the required columns need be set.

- INOUT parameter of a stored procedure:

  The INOUT argument behaves as both an IN and an OUT parameter and therefore follows rules for both IN and OUT parameters. The database manager sets the argument before a stored procedure is called. The *SQL-argument-ind* is set by the stored procedure before it is returned to the database manager.

Each *SQL-argument-ind* value is in form of SMALLINT.

SQL-argument-ind-array
The *SQL-argument-ind-array* argument is array of elements where each element corresponds to each *SQL-argument* value that is passed to a stored procedure. The *n*th element in *SQL-argument-ind-array* argument corresponds to the *n*th *SQL-argument* value and indicates whether the *SQL-argument* argument contains a NULL value.

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

- IN parameter of a stored procedure:

  The IN parameter element is set by the database manager before a routine is called. 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 the `RETURNS NULL ON NULL INPUT` option, the stored procedure body does not need to check for a NULL value. However, if the stored procedure is defined with the `CALLED ON NULL INPUT` option, any argument can be NULL. If your stored procedure is defined with the `CALLED ON NULL INPUT` option, your stored procedure must check the *SQL-argument-ind* value before it uses the corresponding *SQL-argument* value.

- OUT parameter of a stored procedure:

  The OUT parameter element is set by the routine before it is returned to the database manager. The *SQL-argument-ind-array* 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:

  The INOUT parameter element behaves as both an IN and an OUT parameter. The INOUT parameters must follow rules for both IN and OUT parameters. The database manager sets the argument before a stored procedure is called. The element of *SQL-argument-ind-array* is set by the stored procedure before it is returned to the database manager.

Each element of *SQL-argument-ind-array* value is in form of SMALLINT.

sqlstate
The *sqlstate* argument is set by the routine before it is returned to the database manager. The *sqlstate* argument 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* value is in form of CHAR(5).

routine-name
The routine name argument is set by the database manager before a routine is called. It is the qualified function name that is passed from the database manager to a routine.

The *routine-name* is passed in the following format:

```
schema.routine
```

The routine name can contain a schema name and a routine name, which are separated by a period. The following example contains two routine names.

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

You can use the *schema.routine* form to have same routines under a different schema name.

Although it is possible to include the period in object names and schema names, avoid use of the period in object names to avoid confusion. For example, if you have a routine name that is passed as the OBJ.OP.ROTATE value, it is difficult to distinguish the schema name or the routine name.

The *routine-name* value is in form of VARCHAR(257).

specific-name

The *specific-name* argument is set by the database manager before a routine is called. It is the specific name that is passed from the database manager to a routine.

The following lists two examples of the specific name value:

```
WILLIE_FIND_FEB99   SQL9904281052440430
```

The first example of the specific name value is provided by the user in the CREATE statement. The second example of the specific name value is generated by the database manager from the current timestamp when the user does not specify the specific name value.

As with the *routine-name* argument, the specific name value is used to clearly identify the routine.

The *specific-name* value is in form of VARCHAR(18).

diagnostic-message

The *diagnostic-message* argument is set by the routine before it is returned to the database manager. The routine can use the *diagnostic-message* argument to insert a message text in a database message.

Routines can include descriptive information in the *diagnostic-message* argument with the *sqlstate* argument when an error or a warning is encountered. The database manager includes this information as a token in its message.

The database manager sets the first character to null before a routine is called. The diagnostic message that is returned by the routine is treated as a C null-terminated string. The diagnostic message string is included in the SQLCA structure as a token for the error condition. The first part of the diagnostic message string appears in the SQLCA or CLP message. However, the actual number of characters that appear depends on the lengths of the other tokens. The database manager might truncate the tokens to conform to the total token length limit that is imposed by the SQLCA structure. Avoid use of X'FF' in the diagnostic message string. The X'FF' value is used to delimit tokens in the SQLCA structure.

You must ensure that the routine only returns texts that fits in the VARCHAR(70) buffer that is passed to it. An overwrite by the routine can cause unpredictable results or an abend.

The database manager assumes that any message tokens returned from the routine to the database manager are in the same code page as the routine. You must ensure that the code page that is set in the database manager is same as the routine. If you use the 7-bit invariant ASCII subset, your routine can return the message tokens in any code page.

The *diagnostic-message* value is in form of VARCHAR(70).

scratchpad

The *scratchpad* argument is set by the database manager before an UDF or a method is called. It is only present for functions and methods that specified the **SCRATCHPAD** keyword during registration. The *scratchpad* argument is a structure, which is similar to the `sqllob` data structure and contains the following elements:

- An integer field that contains the length of the scratchpad. Changing the length of the scratchpad results in SQLCODE -450 (SQLSTATE 39501)

- The actual scratchpad that is initialized to all zero binary value as listed in the following scenarios:
  - For scalar functions and methods, the scratchpad is initialized before the first call, and not read or modified by the database manager thereafter.
  - For table functions, the scratchpad is initialized prior to the FIRST call to an UDF if the FINAL CALL option is specified in the CREATE FUNCTION statement. After the FIRST call, the scratchpad content is totally under control of the table function. If an option other than FINAL CALL is specified, the scratchpad is initialized for each OPEN call and the scratchpad content is completely under control of the table function between OPEN calls. The scratchpad initialization behavior for a UDF with the FINAL CALL option that is specified in the CREATE function statement is important for a table function that is used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then the FINAL CALL option must be specified in your CREATE FUNCTION statement. When the FINAL CALL option is specified, the table function also receives the FIRST and FINAL calls in addition to the normal OPEN, FETCH, and CLOSE calls for scratchpad maintenance and resource release.

The scratchpad can be mapped in your routine using the same type as either a CLOB or a BLOB, as the argument passed has the same structure.

Ensure that your routine code does not write outside of the scratchpad buffer. An overwrite by the routine can cause unpredictable results that include an abend.

If a scalar UDF or method that uses a scratchpad is referenced in a subquery, the database manager might decide to refresh the scratchpad between invocations of the subquery. If the `FINAL CALL` option is specified for an UDF, the scratchpad is refreshed after a final-call is made.

call-type

The *call-type* argument, if present, is set by the database manager before an UDF or a method is called. The *call-type* argument is present for the all UDFs and methods that are registered with the `FINAL CALL` option.

You can avoid future exceptions that can arise as the result of added new call types if your UDF or method contains a switch or a case statement that explicitly test for all the expected values.

Your routine can set the *sqlstate* and *diagnostic-message* argument for all *call-type* values.

The include file `sqludf.h` is intended for use with routines. The `sqludf.h` file contains symbolic define statements for the following *call-type* values:

The following *call-type* values can be specified for scalar functions and methods:

SQLUDF_FIRST_CALL (-1)

The `SQLUDF_FIRST_CALL` value indicates the FIRST call to the routine for a statement. If any scratchpad is specified with the *scratchpad* argument, the scratchpad is set to binary zeros when the routine is called. All argument values are passed, and the routine performs one-time initialization actions that are required. The routine is expected to return the result with the *sqlstate* and *diagnostic-message* information.

If the *scratchpad* argument is specified without the `FINAL CALL` option in the **CREATE FUNCTION** statement, the routine cannot use the `SQLUDF_FIRST_CALL` call type to identify the first call. A routine that has the *scratchpad* argument that is specified without the `FINAL CALL` option in the **CREATE FUNCTION** statement must rely on all-zero state in the scratchpad to identify the first call.

SQLUDF_NORMAL_CALL (0)

The `SQLUDF_NORMAL_CALL` value indicates a normal call. All the SQL input values are passed, and the routine is expected to return the result. The routine can also return *sqlstate* and *diagnostic-message* information.

SQLUDF_FINAL_CALL (1)

The `SQLUDF_FINAL_CALL` value indicates a final call where 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.

SQLUDF_FINAL_CRA (255)
The SQLUDF_FINAL_CRA value indicates a final call that is identical to the SQLUDF_FINAL_CALL call type, but the SQLUDF_FINAL_CRA call type is made to routines that are defined as being able to issue SQL statements. The SQLUDF_FINAL_CRA call type is made when the routine must not issue any SQL statements other than CLOSE cursor. For example, when the database manager is in the middle of COMMIT processing, it cannot issue any new SQL statement, and any FINAL call that is issued to a routine would be a call with the SQLUDF_FINAL_CRA call type. Routines that are not defined as containing any SQL statements do not receive the SQLUDF_FINAL_CRA call type, whereas routines that contain SQL statements can receive either the SQLUDF_FINAL_CALL or SQLUDF_FINAL_CRA call types.

A scalar UDF or method is expected to release resources that were allocated. For example, the resource can include memory. When the SQLUDF_FINAL_CALL or SQLUDF_FINAL_CRA call type is specified, resources that were allocated for the routine execution are released if the scratchpad is also specified and is used to track the resource. If the FINAL CALL option is not specified with the **CREATE** statement, allocated resources for a routine must be released when the call statement is completed.

The following *call-type* values can be specified for table functions:

SQLUDF_TF_FIRST (-2)
The SQLUDF_TF_FIRST value indicates the first call, which occurs only if the FINAL CALL option was specified for the UDF. The scratchpad is set to binary zeros before a call is made with the SQLUDF_TF_FIRST call type. All argument values are passed to the table function. The table function can acquire memory or perform other one-time only resource initialization. The call with the SQLUDF_TF_FIRST call type is not an open call. An open call follows this call. On a first call, the table function must not return any data to the database manager as the database manager ignores the data.

SQLUDF_TF_OPEN (-1)
The SQLUDF_TF_OPEN value indicates the open call. The scratchpad is initialized if the NO FINAL CALL option is specified, but not necessarily otherwise. All SQL argument values are passed to the table function. The table function must not return any data to the database manager on the open call.

SQLUDF_TF_FETCH (0)
The SQLUDF_TF_FETCH value indicates a fetch call, and the database manager expect the table function to return either of the following values:

- A row that contains the set of return values.
- An end-of-table condition value that is indicated by SQLSTATE 02000.

If a *scratchpad* is also passed, it is untouched from the previous call.

SQLUDF_TF_CLOSE (1)
The SQLUDF_TF_CLOSE value indicates a close call to the table function. The SQLUDF_TF_CLOSE call type can be specified to close external resources such as source file, and release resources when the NO FINAL CALL option is specified.

In cases that involve a join or a subquery, the open, fetches, and close call sequences can repeat within the execution of a statement. However, there can be only one first call and only one final call. The first and final calls occur only when the FINAL CALL option is specified for the table function.

SQLUDF_TF_FINAL (2)
The SQLUDF_TF_FINAL value indicates a final call. The SQLUDF_TF_FINAL call type is specified only when the FINAL CALL option was specified for the table function. The SQLUDF_TF_FINAL call type can occur only once per execution of a statement. It is intended for releasing resources.

SQLUDF_TF_FINAL_CRA (255)
The SQLUDF_TF_FINAL_CRA value indicates a final call that is identical to the SQLUDF_TF_FINAL call type, but the SQLUDF_TF_FINAL_CRA call type is made to UDFs that

are defined as being able to SQL statements. The SQLUDF_TF_FINAL_CRA call type is made when the UDF must not issue any SQL statements other than CLOSE cursor. For example, when the database manager is in the middle of COMMIT processing, it cannot issue any new SQL statement, and any FINAL call that is issued to a UDF would be a call with the SQLUDF_TF_FINAL_CRA call type. UDFs that are not defined as containing any SQL statements do not receive the SQLUDF_TF_FINAL_CRA call type, whereas UDFs that contain SQL statements can receive either the SQLUDF_TF_FINAL or SQLUDF_TF_FINAL_CRA call types.

You must ensure that your routine releases all resources it allocates. For table functions, you can release the resources on the close call and the final call. The close call can occur multiple times in the execution of a statement while the final call can occur only once in a statement if the FINAL CALL option was specified. You can optimize the resource usage by allocating the resource during a call with the SQLUDF_TF_FIRST call type and releasing the resources on a call with the SQLUDF_TF_FINAL or SQLUDF_TF_FINAL_CRA call type. If the FINAL CALL option is specified, the scratchpad is initialized only before a call with the SQLUDF_TF_FIRST call type. If a call with the SQLUDF_TF_FINAL or SQLUDF_TF_FINAL_CRA call type is not specified, then the scratchpad is reinitialized before each call with the SQLUDF_TF_OPEN call type.

The *call-type* takes the form of an integer value.

dbinfo

The **dbinfo** argument is set by the database manager before a routine is called. It is only present if the **CREATE** statement for the routine specifies the **dbinfo** keyword. The sqludf_dbinfo structure that is defined in the header file sqludf.h is passed as the **dbinfo** argument. The sqludf_dbinfo structure contains variable names and identifiers that can be longer than the maximum value, which is supported by the database manager. The longer variable length is designed 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 used. The sqludf_dbinfo structure contains the following elements:

1. Database name length (**dbnamelen**)

   The length of a database name. 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 **dbnamelen** field 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. This field is an unsigned short integer.

4. Application authorization ID (**authid**)

   The application runtime authorization ID. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The **authidlen** field identifies the actual length of this field.

5. Environment code pages (**codepg**)

   The code page information.

   The environment code page is a union of three 48-byte structures:

   - The **cdpg_db2** structure that is common to all Db2 products.
   - The **cdpg_cs** structure that represents older version of databases that are now deprecated.
   - The **cdpg_mvs** structure that represents older legacy databases that are now deprecated.

   You can use the **cdpg_db2** structure for the **codepg** element. The **cdpg_db2** structure is made up of an array (**db2_ccsids_triplet**) of three sets of code page information that represents the possible encoding schemes in the database as follows:

a. ASCII encoding scheme. For compatibility with previous version of IBM database, if the database is a Unicode database then the information for the Unicode encoding scheme is placed here and 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. 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. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The **tbschemalen** field identifies the actual length of this field.

8. Table name length (**tbnamelen**)

   The length of the table name. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

9. Table name (**tbname**)

   Name of the table that being updated or inserted. The **tbname** 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 identifies the actual length of this field. The schema name field with the table name field forms the fully qualified table name.

10. Column name length (**colnamelen**)

    Length of column name. It contains a 0 (zero) if a column name is not passed. This field is an unsigned short integer.

11. Column name (**colname**)

    Name of the column that is being updated or inserted. The **colname** field can contain up to 128 characters. It does not contain a null terminator or any padding. The **colnamelen** field identifies the actual length of this field.

12. Version/Release number (**ver_rel**)

    An eight character field that identifies the product and its version, release, and modification level with the format *pppvvrrm* where:

    - *ppp* identifies the product as follows:

      DSN
      Db2 for z/OS or OS/390®

      ARI
      SQL/DS or Db2 for VM or VSE

      QSQ
      Db2 for IBM i

      SQL
      Db2

    - *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**)

    The reserved field is for future use.

14. Platform (**platform**)

    The operating system (**platform**) for the application server, as follows:

    SQLUDF_PLATFORM_AIX
    AIX®

    SQLUDF_PLATFORM_LINUX
    Linux

    SQLUDF_PLATFORM_ZOS
    Db2 for z/OS

    SQLUDF_PLATFORM_WINDOWS
    Windows operating systems

    SQLUDF_PLATFORM_UNKNOWN
    Unknown operating system or platform

    For operating systems that are not contained in the preceding list, see the contents of the `sqludf.h` file.

15. Number of table function column list entries (**numtfcol**)

    The number of column entries in the result column list of the table function.

16. Reserved field (**resd1**)

    The reserved 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 the SYSCAT.ROUTINES catalog view, 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 the routine is a table function, the **tfcolumn** field is a pointer to an array of short integers that is dynamically allocated by the database manager. If a routine is not a table function, the **tfcolumn** pointer is null.

    The **tfcolumn** field is used only for table functions. Only the first *n* entries, where *n* is a value that is specified in the **numtfcol** field, are relevant. The *n* value must be equal or greater than 0, and it must be equal or less than the number of result columns that are defined for the RETURNS TABLE(...) clause in the **CREATE FUNCTION** statement. Each element value in the array corresponds to an ordinal number of a result column from the table function. Order of the array elements is arranged in the order that is required by the statement. For example, a value of '1' means the first defined result column, '2' means the second defined result column. The ordinal number value of a result column can be in any order. If no actual result column values from the table function are needed by the statement, the *n* value and the `numtfcol` variable value is zero.

    The use of the **tfcolumn** field optimizes the performance since UDF is not required to return all the result columns from the table function.

20. Unique application identifier (**appl_id**)

    The unique application identifier field is a pointer to a C null-terminated string that uniquely identifies the application's connection to a database. It is generated by the database manager at connect time.

The string has a maximum length of 32 characters in the following format:

```
x.y.ts
```

where the *x* and *y* values vary by connection type, but the *ts* value represents a 12 character time stamp value of the form YYMMDDHHMMSS. The *ts* value ensures uniqueness of the string.

```
Example:  *LOCAL.db2inst.980707130144
```

21. Reserved field (**resd3**)

    This field is for future use.

Graphic host variables in C and C++ routines

Routines 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.

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. C++ compilers type-decorate or 'mangle' function names by default. The type-decorated name consists of argument type names that are appended to their function names.

The names of C++ functions can be overloaded. Two C++ functions with the same name can coexist if they have different arguments, for example the first `func` function takes an integer type argument while second `func` function takes character type argument:

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

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

C++ compilers type-decorate or 'mangle' function names by default, where the argument type names are appended to their function names, as in `func__Fi` and `func__Fc` for the two earlier examples. The mangled names are 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, with the `nm` command. The **nm** 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 that is produced by all of these commands includes a line with the mangled function name. The following **nm** command output example contains the mangled function name:

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

Once you obtained the mangled function name from one of the preceding commands, you can use it in the appropriate command.

When you register a routine with the **CREATE** statement, the EXTERNAL NAME clause must specify the mangled function name. The following example registers the mangled function name with use of the EXTERNAL NAME clause:

```
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. You can always overload the SQL function names that are given to UDFs since the database manager resolves what library function to call based on the function name and the parameter 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 */
```

The `fold` and `findvwl` UDFs are not type-decorated by the compiler, and they must be registered in the **CREATE FUNCTION** statement with 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.

Returning result sets from C and C++ procedures
You can develop C and C++ procedures that return result sets to a calling routine or application that is implemented using an API that supports the retrieval of procedure result sets.

Most APIs support the retrieval of procedure result sets, however embedded SQL does not.

The C and C++ representation of a result set is an SQL cursor. Any SQL cursor that has been declared, opened, and not explicitly closed within a procedure, prior to the return of the procedure can be returned to the caller. The order in which result sets are returned to the caller is the same as the order in which cursor objects are opened within the routine. No additional parameters are required in the CREATE PROCEDURE statement or in the procedure implementation in order to return a result set.

Prerequisites

A general understanding of how to create C and C++ routines will help you to perform the steps in the following procedure for returning results from a C or C++ procedure.

“Creating .NET CLR routines from the Db2 command window” on page 83Creating C and C++ routines

Cursors declared in C or C++ embedded SQL procedures are not scrollable cursors.

Procedure

To return a result set from a C or C++ procedure:

1. In the CREATE PROCEDURE statement for the C or C++ procedure you must specify along with any other appropriate clauses, the DYNAMIC RESULT SETS clause with a value equal to the maximum 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 C or C++ procedure implementation of your routine, declare a cursor using the DECLARE CURSOR statement within the declaration section in which host variables are declared. The cursor declaration associates an SQL with the cursor.
4. Within the C or C++ routine code, open the cursor by executing the OPEN statement. This executes the query specified in the DECLARE CURSOR statement and associates the result of the query with the cursor.
5. Optional: Fetch rows in the result set associated with the cursor using the FETCH statement.
6. Do not execute the CLOSE statement used for closing the cursor at any point prior to the procedure's return to the caller. The open cursor will be returned as a result set to the caller when the procedure returns.

   When more than one cursor is left open upon the return of a procedure, the result sets associated with the cursors are returned to the caller in the order in which they were opened. No more than the maximum number of result sets specified by the DYNAMIC RESULT SETS clause value can be returned with the procedure. If the number of cursors left open in the procedure implementation is greater than the value specified by the DYNAMIC RESULT SETS clause, the excess result sets are simply not returned. No error or warning will be raised by the database manager in this situation.

Once the creation of the C or C++ procedure is completed successfully, you can invoke the procedure with the CALL statement from the Db2 Command Line Processor or a Db2 Command Window to verify that the result sets are successfully being returned to the caller.

For information on calling procedures and other types of routines:

- Routine invocation

Creating C and C++ routines
Procedures and functions that reference a C or C++ library are created in a similar way to external routines with other implementations. This task comprises a few steps including the formulation of the CREATE statement for the routine, the coding of the routine implementation, precompilation, compilation and linking of code, and the deployment of source code.

Before you begin

- Knowledge of C and C++ routine implementation. To learn about C and C++ routines in general see:
  - "C and C++ routines" on page 118
- The IBM data server client which includes application development support must be installed on the client computer.
- The database server must be running an operating system that supports a C or C++ compiler that is supported by the database.
- The required compilers must be installed on the database server.
- Authority to execute the CREATE statement for the external routine. For the privileges required to execute the CREATE PROCEDURE statement or the CREATE FUNCTION statement, see the documentation for the statement.

About this task

You would choose to implement a C or C++ routine 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 using an embedded SQL programming language such as C or C++.

Procedure

1. Code the routine logic in the chosen programming language: C or C++.
   - Include any C or C++ header files that are required for additional C functionality and the database header files for C or C++ that are required for SQL data type and SQL execution support. Include the following header files:
     - `memory.h`
     - `sql.h`
     - `sqlca.h`
     - `sqlda.h`
     - `sqludf.h`
   - A routine parameter signature must be implemented using one of the supported parameter styles. It is strongly recommended that parameter style SQL be used for all C and C++ routines. Scratchpads and `dbinfo` structures are passed into C and C++ routines as parameters. For more on parameter signatures and parameter implementations see:
     - "Parameters in C and C++ routines" on page 121
     - "Parameter style SQL C and C++ procedures" on page 122
     - "Parameter style SQL C and C++ functions" on page 125
   - Declare host variables and parameter markers in the same manner as is done for embedded SQL C and C++ applications. Be careful to correctly use data types that map to SQL data types. For more on data type mapping between SQL and C or C++ data types refer to:
     - "Supported SQL data types in C and C++ routines" on page 131
   - Include routine logic. Routine logic can consist of any code supported in the C or C++ programming language. It can also include the execution of embedded SQL statements which is implemented in the same way as for embedded SQL applications.

     For more on executing SQL statements in embedded SQL see:

  - "Executing SQL statements in embedded SQL applications" in *Developing Embedded SQL Applications*
- 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 routines:
  - “Returning result sets from C and C++ procedures” on page 155
- Set a routine return value at the end of the routine.

2. Build your code to produce a library file.

   For information on how to build embedded SQL C and C++ routines, see:

   - “Building C and C++ routine code” on page 157
3. Copy the library into the database *function directory* on the database server. It is recommended that you store libraries associated with 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 library to another directory on the server, but to successfully invoke the routine you must note the fully qualified path name of your library 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: C
   - Specify the PARAMETER STYLE clause with the name of the supported parameter style that was implemented in the routine code. It is strongly recommended that PARAMETER STYLE SQL be used.
   - Specify the EXTERNAL clause with the name of the library to be associated with the routine using one of the following values:
     - the fully qualified path name of the routine library
     - the relative path name of the routine library relative to the function directory.

     By default the database manager looks for the library in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.
   - Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure and it will return one or more result sets to the caller.
   - Specify any other non-default clause values in the CREATE statement to be used to characterize the routine.

Results

To invoke your C or C++ routine, see “Invoking routines” on page 205.

Building C and C++ routine code

Once embedded SQL C or C++ routine implementation code is written, it must be built into a library and deployed before the routine can be called. Although the steps required to build embedded SQL C and C++ routines are similar to those steps that are required to build embedded SQL C and C++ applications, there are some differences.

Procedure

There are two ways to build C and C++ routines:

- Using database sample build scripts (Linux and UNIX) or build batch files (Windows)
- Entering database and C or C++ compiler commands from the Db2 Command Window

The database sample build scripts and batch files are designed for building database sample routines (procedures and user-defined functions) and user created routines for a particular operating system that uses the default supported compilers.

There is a separate set of database sample build scripts and batch files for C and C++. In general, it is easier to build embedded SQL routines with the build scripts or batch files, which can easily be modified

as required. However, it is often helpful to know the database and C or C++ compiler commands to build routines from Db2 Command Window.

Building C and C++ routine code using sample bldrtn scripts

Building C and C++ routine source code is a subtask of creating C and C++ routines. You can easily build C and C++ routines with the database sample build scripts (Linux and UNIX) and batch files (Windows). The sample build scripts can be used for source code with or without embedded SQL statements. The build scripts can pre-compile, compile, and link C or C++ source code. The build scripts can also bind any packages that are associated with the embedded SQL routine to the specified database.

Before you begin

The sample build-script for building C and C++ routines is named `bldrtn`. The `bldrtn` script is located in the following database directories along with the sample programs:

- For C:`sqllib/samples/c/`
- For C++: `sqllib/samples/cpp/`

The `bldrtn` script can be used to build a source code file that contains both procedure and function implementations. The `bldrtn` script performs the following tasks:

- Establishes a connection with a user-specified database
- Precompiles the user-specified source code file
- Binds the package to the current database
- Compiles and links the source code to generate a shared library
- Copies the shared library to the database function directory on the database server

The `bldrtn` scripts accept two arguments:

- The name of a source code file without any file extension
- The name of a database to which a connection is established

The database parameter is optional. If no database name is supplied, the program uses the default sample database. As routines must be built on the same instance where the database resides, no arguments are required for a user ID and password.

- Source code file that contains one or more routine implementations.
- The name of the database within the current database instance in which the routines are to be created.

Procedure

To build a source code file that contains one or more routine code implementations:

1. Open the Db2 command window.
2. Copy your source code file into the same directory as the **bldrtn** script file.
3. If the routines will be created in the sample database, enter the build script name followed by the name of the source code file without the `.sqc` or `.sqC` file extension:

   ```
   bldrtn file-name
   ```

   If the routines will be created in another database, enter the build script name, the source code file name without any file extension, and the database name:

   ```
   bldrtn file-name database-name
   ```

   The `bldrtn` script precompiles, compiles, and links the source code and produces a shared library. The script then copies the shared library to the function directory on the database server.
4. If this is not the first time that the source code is built, stop and restart the database instance to ensure that the new version of the shared library is used by the database. You can restart the database instance by entering the **db2stop** command followed by the **db2start** command.

What to do next

Once you have successfully built the routine shared library and deployed it to the function directory on the database server, you should complete the steps associated with the task of creating C and C++ routines. After routine creation is completed you will be able to invoke your routines.

Building C and C++ routine code from the Db2 command window

You can build C and C++ routines from the command line. The procedure to build C and C++ routines include compilation and linking of the source code, followed by deploying the routine library. If you have any embedded SQL statements present in the source code, you must first run the precompiler before you can proceed with the C or C++ routine compilation. You must also bind the resulting bind file that is generated by the precompiler.

Before you begin

As an alternative to manually building the C and C++ routines, you can modify the sample build scripts that are provided in the `sample` subdirectory of the `sqllib` directory to simplify this task.

The following conditions must be met before you can build the C and C++ source code with embedded SQL statements:

- You must have a source code file that contains one or more embedded SQL C or C++ routine implementations.
- You must have the database information in which the routines are to be created.
- You must have compiler options that are operating system specific and link options that are required for building C and C++ routines.

Procedure

To build a source code file that contains one or more embedded SQL statements:

1. Open the Db2 command window.
2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database in which the routines are created.
4. Precompile the source code file.
5. Bind the package that was generated to the database.
6. Compile the source code file.
7. Link the source code file to generate a shared library. The linking process requires the use of compiler link options that references the database `include` directory.
8. Copy the shared library to the database function directory on the database server.
9. If you are not building the source code file for the first time, you must stop and restart the database manager to ensure that the new version of the shared library is used by the database manager. You can issue the **db2stop** command followed by the **db2start** command to restart the database manager.

Results

Once you successfully built and deployed the routine library, you can register the new routine in the database by running the **CREATE** statement. The routine must be registered before you can call the new routine.

Example

The following example builds an embedded SQL C++ source code file that is named `myfile.sqC`, which contains a routine implementation. The routine is compiled with IBM C and C++ compiler on the AIX operating system.

1. Open the Db2 command window.

2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database to which the routine is intended for.

```
db2 connect to database-name
```

4. Precompile the source code file with the **PREPARE** or **PREP** command.

```
db2 prep myfile.sqC bindfile
```

   The precompiler displays an output to indicate whether there were any errors. The precompile step generates a bind file that is named `myfile.bnd`, which can be used to generate a package in the next step.
5. Bind the package that was generated to the database with the **BIND** command.

```
db2 bind myfile.bnd
```

   The bind utility displays an output to indicate whether there were any errors.
6. Compile the source code file and specify any compiler options that you need to specify. You must include the database `include` directory.

```
xlC_r -qstaticinline -I$HOME/sqllib/include -c $myfile.C
```

   The compiler displays an output to indicate whether there are any errors. This step generates an export file named `myfile.exp`.
7. Link the compiled object to generate a shared library.

```
xlC_r -qmkshrobj -o $1 $1.o -L$ HOME/sqllib/include/lib32 -lDB2
```

   The linker displays an output to indicate whether there are any errors. This step generates a shared library file name `myfile`.
8. Copy the shared library to the database `function` directory on the database server.

```
rm -f ~HOME/sqllib/function/myfile
cp myfile $HOME/sqllib/function/myfile
```

   This step ensures that the routine library is in the default directory where the database manager looks for routine libraries.
9. Stop and restart the database manager.

```
db2stop
db2start
```

The **LD_LIBRARY_PATH** operating system environment variable is ignored when a source code with embedded SQL statements is built.

Compile and link options for C and C++ routines

Rebuilding routine shared libraries

The database manager caches the shared libraries used for stored procedures and user-defined functions once loaded. If you are developing a routine, you might want to test loading the same shared library a number of times, and this caching can prevent you from picking up the latest version of a shared library. The way to avoid caching problems depends on the type of routine.

1. **Fenced, not threadsafe routines.** The database manager configuration keyword **keepfenced** has a default value of YES. This keeps the fenced mode process alive. This default setting can interfere with reloading the library. It is best to change the value of this keyword to NO while developing fenced, not threadsafe routines, and then change it back to YES when you are ready to load the final version of your shared library. For more information, see “Updating the database manager configuration parameters” on page 161.

2. **Trusted or threadsafe routines.** Except for SQL routines (including SQL procedures), the only way to ensure that an updated version of a routine library is picked up when that library is used for trusted, or threadsafe routines, is to recycle the database instance by entering **db2stop** followed by **db2start** on the command line. This is not needed for an SQL routine because when it is recreated, the compiler uses a new unique library name to prevent possible conflicts.

For routines other than SQL routines, you can also avoid caching problems by creating the new version of the routine with a differently named library (for example foo.a becomes foo.1.a), and then using either the ALTER PROCEDURE or ALTER FUNCTION SQL statement with the new library.

Updating the database manager configuration parameters

You can configure the database manager configuration parameters for your stored procedures and UDFs. The two key database manager configuration parameters are **KEEPFENCED** and **JDK_PATH**.

About this task

The database manager configuration parameter **KEEPFENCED** has the default value YES. When you are developing new routines (stored procedures and UDFs), set the **KEEPFENCED** parameter to NO. Setting the **KEEPFENCED** parameter to NO avoids routine process from persisting in the system and ensure that the routine that is running on the system is the current version. You can change the **KEEPFENCED** parameter back to YES when you deploy the final version of your routine.

For Java routines, the **JDK_PATH** parameter must be set.

Procedure

To change the database manager configuration parameter settings, enter the following command:

```
db2 update dbm cfg using parameter value
```

For example, enter the following command to set the **KEEPFENCED** parameter to NO:

```
db2 update dbm cfg using KEEPFENCED NO
```

To set the **JDK_PATH** parameter to the directory /home/db2inst/jdk17, enter the following command:

```
db2 update dbm cfg using JDK_PATH /home/db2inst/jdk17
```

Results

To view the current settings in the database manager configuration parameters, enter the following command:

```
db2 get dbm cfg
```

Note: On Windows operating system, you need to enter the **db2 update dbm cfg** command in the Db2 command window.

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'
```

Support for external procedure development in COBOL
To develop external procedures in COBOL you must use the supported COBOL development software.

All of the development software supported for database application development in COBOL can also be used for external procedure development in COBOL.

Building COBOL routines

Compile and link options for COBOL routines

Java routines

Java routines are external routines that have a Java programming language implementation.

Java routines are created in a database by executing a CREATE PROCEDURE or CREATE FUNCTION statement. This statement must indicate that the routine is implemented in Java with the LANGUAGE JAVA clause. It must also specify with the EXTERNAL clause, the Java class that implements it.

External procedures, functions, and methods can be created in Java.

Java routines can execute SQL statements.

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

CREATE statement
The SQL language CREATE statement used to create the routine in the database.

Routine-body source code
The source code file containing the Java routine implementation. The Java routine can access the database using either JDBC or SQLJ application programming interfaces.

JDBC
Application programming interface that provides support for dynamic SQL statement execution in Java code.

SQLJ
Application programming interface that provides support for static SQL statement execution in Java code.

SDK for Java
Software development kit for Java provided and required for Java source code compilation.

Routine class
A Java source code file containing the compiled form of Java routine source code. Java class files can exist on their own or they can be one of a collection of Java class files within a JAR file.

Supported Java routine development software

To develop and deploy external routines in Java, you must use supported Java development software.

The minimum supported software development kit (SDK) version for Java routine development is the IBM SDK for Java 1.4.2. However, support for the IBM SDK for Java 1.4.2 is deprecated, and might be discontinued in a future release.

The maximum supported SDK version for Java routine development is IBM SDK for Java 8. IBM SDK for Java 8 supports stored procedures and user-defined functions on the following operating systems:

- AIX
- Linux on x86
- Linux on AMD64/EM64T
- Linux on POWER®
- Windows on x86
- Windows on x64, for AMD64/EM64T

It is recommended that you use the SDK for Java that is installed with the database product, however an alternative SDK for Java can be specified. If you specify an alternative SDK for Java, it must be of the same bit-width as the database instance.

All supported software development tools for Java can be used to develop Java external routines.

JDBC and SQLJ application programming interface support for Java routines

External routines that are developed in Java can use the following application programming interfaces (APIs): JDBC and SQLJ.

The IBM Data Server Driver for JDBC and SQLJ product supports both the JDBC and SQLJ APIs and can be used to develop external Java routines.

The procedures for implementing Java routines are the same regardless which API is used.

Specifying JDK for Java routine development (Linux and UNIX)

To build and run Java routine code, you must set the **JDK_PATH** database manager configuration parameter to the Java Development Kit (JDK) installation path on the database server.

Before you begin

The IBM database server products and IBM data server client products installs the JDK by default and the **JDK_PATH** database manager configuration parameter is set to the *$INSTDIR*/sqllib/java/jdk64 path.

The **JDK_PATH** parameter value can be changed to specify another JDK that is installed on the computer. However, the JDK must be of the same bit-width as the IBM data server product that is installed.

The following conditions must be met before you can set the **JDK_PATH** database manager configuration parameter:

- User must have access to the database server.

- User must have the authority to read and update the database manager configuration file.
- User must have authority to install the JDK in a file system where the IBM data server product is installed.

Procedure

1. Check the **JDK_PATH** parameter value with the **get dbm cfg** command from the Db2 command window:

   ```
   db2 get dbm cfg
   ```

   You might want to redirect the output to a file for easier viewing.

   The **JDK_PATH** parameter value appears near the beginning of the output.
2. If you want to use a different JDK, install the JDK on the database server and note the JDK installation path.
3. Update the **JDK_PATH** parameter value with the **update dbm cfg** command from the Db2 command window, where *path* is the path where the new JDK is installed:

   ```
   db2 update dbm cfg using JDK_PATH path
   ```
4. Stop and restart the database manager:

   ```
   db2stop

   db2start
   ```
5. Verify the **JDK_PATH** parameter value by running the following command:

   ```
   db2 get dbm cfg
   ```

What to do next

After you complete these steps, the JDK that is specified with the **JDK_PATH** parameter is used to run Java routines. The **CLASSPATH**, **PATH**, and **LIBPATH** environment variables for JDK are set automatically when the **JDK_PATH** parameter is set.

Specification of a driver for Java routines
Java routine development and invocation requires that a JDBC or SQLJ driver be specified.

Java routines use the IBM Data Server Driver for JDBC and SQLJ Version 4.0.

IBM Data Server Driver for JDBC and SQLJ Version 4.0 `db2jcc4.jar` includes a number of JDBC Version 4.0 capabilities.

By default, Java routines use the IBM Data Server Driver for JDBC and SQLJ to access databases. The following features are supported in the Java routine with the IBM Data Server Driver for JDBC and SQLJ Version 4.0:

- Parameters of data type XML
- Variables of data type XML
- References to XML data
- References to XML functions
- Any other native-XML feature

Tools for developing Java (JDBC and SQLJ) routines
Tools such as IBM Data Studio, the Db2Command Line Processor and the Db2Command Window make Java routine development go quickly and easily.

The following database tool provides graphical user-interface support for developing, debugging, and deploying Java routines:

- IBM Data Studio

The following command-line interfaces can also be used for developing, debugging, and deploying Java routines:

- Db2Command Line Processor
- Db2Command Window

Other IBM software products provide graphical tools for developing Java routines including:

- IBM IBM Data Studio
- IBM Rational Application Developer
- Distributed Unified Debugger

Designing Java routines
Designing Java routines is a task that should precede creating Java routines. Designing Java routines is related to both designing external routines implemented in other programming languages and designing Java database applications.

Before you begin

Knowledge and experience with embedded SQL application development as well as general knowledge of external routines. The following topics can provide you with some of the required prerequisite information.

For more information on the features and uses of external routines, see:

- “External routine implementation” on page 18

For more information about how to program a basic Java application using either the JDBC or SQLJ application programming interfaces, see:

- "Example of a simple JDBC application" in *Developing Java Applications*
- "Example of a simple SQLJ application" in *Developing Java Applications*

About this task
With the prerequisite knowledge, designing Java routines consists mainly of learning about the unique features and characteristics of Java routines:

Procedure

- “Supported SQL data types in Java routines” on page 166
- “Parameters in Java routines” on page 169
- “Parameter style JAVA procedures” on page 169
- “Parameter style JAVA Java functions and methods” on page 170
- “Returning result sets from JDBC procedures” on page 179
- “Returning result sets from SQLJ procedures” on page 180
- “Restrictions on Java routines” on page 183
- “Table function execution model for Java” on page 60

What to do next
After having learned about the Java characteristics, you might want to refer to:

- “Creating Java routines from the command line” on page 185

Supported SQL data types in Java routines
Java programming language data types must be used in Java source code to store SQL data type values according to the JDBC and SQLJ application programming interface specification. IBM Data Server Driver

for JDBC and SQLJ converts the data exchanged between Java source code and a database according to specific data type mappings.

The data mappings are valid for:

- Java database applications
- Java routines defined as and implemented using PARAMETER STYLE JAVA

The Java data types that map to SQL data types are as follows:

Table 10. SQL Data Types Mapped to Java Declarations

SQL Column Type	Java Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short	16-bit, signed integer
INTEGER (496 or 497)	int	32-bit, signed integer
BIGINT (492 or 493)	long	64-bit, signed integer
REAL (480 or 481)	float	Single precision floating point
DOUBLE (480 or 481)	double	Double precision floating point
DECIMAL(*p*,*s*) (484 or 485)	java.math.BigDecimal	Packed decimal
CHAR(*n*) (452 or 453)	java.lang.String	Fixed-length character string of length *n* where *n* is from 1 to 255
CHAR(*n*) FOR BIT DATA	byte[]	Fixed-length character string of length *n* where *n* is from 1 to 255
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
BINARY	byte[]	Fixed-length binary string of length *n* where *n* is from 1 to 255
VARBINARY	byte[]	Variable-length binary string
BLOB(*n*) (404 or 405)	java.sql.Blob	Large object variable-length binary string
CLOB(*n*) (408 or 409)	java.sql.Clob	Large object variable-length character string
DBCLOB(*n*) (412 or 413)	java.sql.Clob	Large object variable-length double-byte character string
DATE (384 or 385)	java.sql.Date	10-byte character string

Table 10. SQL Data Types Mapped to Java Declarations (continued)		
SQL Column Type	**Java Data Type**	**SQL Column Type Description**
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	The character string can be from 19 - 32 bytes in length depending on the number of fractional seconds specified. The fractional seconds of the TIMESTAMP data type can be optionally specified with 0-12 digits of timestamp precision. When a timestamp value is assigned to a timestamp variable with a different number of fractional seconds, the value is either truncated or padded with zeros to match the format of the variable.
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
XML(*n*) (408 or 409)	java.sql.Clob	The XML data type is represented in the same way as a CLOB data type; that is as a large object variable-length character string
ARRAY	java.sql.Array	An array of SQL data.
BOOLEAN (2436 or 2437)	boolean	A one byte value that represents a truth value (that is, TRUE or FALSE).

Note:

1. Parameters of an SQL array data type are mapped to class com.ibm.db2.ARRAY.
2. LONG VARCHAR, LONG VARGRAPHIC, XML, REFERENCE, UDT and ARRAY are not supported for the ARRAY data type.

Connection contexts in SQLJ routines

Each SQL statement in the SQLJ routines must explicitly indicate the ConnectionContext object, and that context must be explicitly instantiated in the Java method.

In the following SQLJ example, use of the default context causes all threads to use the same connection context that can result in unexpected failures.

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

```
    }
  }
```

In the following SQLJ example, each invocation of the routine creates its own unique ConnectionContext object (and underlying JDBC connection), which avoids unexpected interference by concurrent threads.

```
#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();
  }
}
```

Parameters in Java routines

Parameter declaration in Java routines must conform to the requirements of one of the supported parameter styles.

The following parameter styles are supported for Java routines:

- PARAMETER STYLE JAVA
- PARAMETER STYLE DB2GENERAL

It is strongly recommended that you specify the PARAMETER STYLE JAVA clause in the routine CREATE statement. With PARAMETER STYLE JAVA, a routine uses a parameter passing convention that conforms to the Java language and SQLJ Routines specification.

The following list contains the Java routine features that cannot be implemented or used with PARAMETER STYLE JAVA:

- Table functions
- Scratchpads in functions
- Access to the DBINFO structure in functions
- The ability to make a FINAL CALL (and a separate first call) to a function or method

You can implement features that cannot be implemented or used with PARAMETER STYLE JAVA by creating your routine in C, or parameter style DB2GENERAL.

Parameter style JAVA procedures

The recommended parameter style for Java procedure implementations is PARAMETER STYLE JAVA.

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`, located in a JAR file that has been cataloged to the database with the Jar ID `myjar`

Note:

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.

Parameter style JAVA Java functions and methods
The recommended parameter style for Java functions and methods is PARAMETER STYLE JAVA.

The signature of PARAMETER STYLE JAVA functions 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 function 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 scalar function is as follows:

```
CREATE FUNCTION product( in1 DOUBLE, in2 DOUBLE )
  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` that is located in a JAR file that has been installed to the database server with the Jar ID `myjar`. JAR files can be installed to a database server using the `INSTALL_JAR` built-in procedure.

DB2GENERAL routines
PARAMETER STYLE DB2GENERAL routines are written in Java. Creating DB2GENERAL routines is very similar to creating routines in other supported programming languages. After you created and registered the routines, 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 these features, it is recommended that you port them to PARAMETER STYLE JAVA.

DB2GENERAL UDFs
You can create and use UDFs in Java just as you would in other languages. 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 is 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 a 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, the database manager 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 that represent 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 this 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, the `int result` variable has an index of 4; in the second, `result1` through `result3` have indices of 2 through 4.

Like C modules that are 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.

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 that is specified in the database manager's **CLASSPATH**.

Typically, the database manager calls a UDF many times. The UDF can be called 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, the database manager 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 database 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. 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, the database manager might elect to evaluate the UDF in parallel. If the database manager elects to evaluate the UDF in parallel, several distinct Java objects can 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, it is suggested that you run UDFs as FENCED UDFs.

You can define an external Java table function that can output tables of different schemas based on input arguments to the function. An external Java table function that can output tables of different schema is known as a generic table function. To define an external Java generic table function, specify PARAMETER STYLE DB2GENERAL in the CREATE FUNCTION statement and declare the shape of the output table function when the external generic table function is referenced.

Supported SQL data types in DB2GENERAL routines
When you call PARAMETER STYLE DB2GENERAL routines, SQL types are converted to and from Java types.

Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 11. SQL Types and Java Objects

SQL Column Type	**Java Data Type**
SMALLINT	short
INTEGER	int
BIGINT	long
REAL"1" on page 174	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" on page 174	String
BINARY	COM.ibm.db2.app.Blob
VARBINARY	COM.ibm.db2.app.Blob
BLOB(*n*)"2" on page 174	COM.ibm.db2.app.Blob

Table 11. SQL Types and Java Objects (continued)	
SQL Column Type	**Java Data Type**
CLOB(*n*)"2" on page 174	COM.ibm.db2.app.Clob
DBCLOB(*n*)"2" on page 174	COM.ibm.db2.app.Clob
DATE"3" on page 174	String
TIME"3" on page 174	String
TIMESTAMP"3" on page 174	String

Note:

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.

Java classes for DB2GENERAL routines
You can use the Java classes for DB2GENERAL routines interface to provide a public java.sql.Connection getConnection() routine to fetch a JDBC connection to the embedding application context.

You can use the `public java.sql.Connection getConnection()` handle to run SQL statements. Other methods of the `StoredProc` interface are listed in the file `sqllib/samples/java/StoredProc.java`.

There are five classes/interfaces that you can use with Java Stored Procedures or UDFs:

- COM.ibm.db2.app.StoredProc
- COM.ibm.db2.app.UDF
- COM.ibm.db2.app.Lob
- COM.ibm.db2.app.Blob
- COM.ibm.db2.app.Clob

The COM.ibm.db2.app.UDF class supports external Java generic table UDFs.

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.

DB2GENERAL Java class: COM.ibm.db2.app.UDF
A Java class that contains methods intended to be called as UDFs with a DB2GENERAL parameter style must be public and must implement the COM.IBM.db2.app.UDF Java interface.

You must declare such a class as follows:

```
public class user-UDF-class extends COM.ibm.db2.app.UDF{ ... }
```

You can call methods of the `COM.ibm.db2.app.UDF` interface only in the context of the currently executing UDF. For example, you cannot use operations on LOB arguments, result or status-setting calls, and so on, after a UDF returns from execution. A Java exception is thrown if this rule is violated.

Argument-related calls use a column index to identify the column being set. The column indexes 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-operation stub handles and ignores 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 must 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 must 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 can call this function with "02000" to signal the end-of-table condition. If the string is not acceptable as an SQLSTATE, an exception is thrown.

```
public void setSQLmessage(String) throws Exception
```

This function is like the `setSQLstate` function. It sets the SQL message result. If the string is not acceptable (for example, longer than 70 characters), an exception is 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 called this routine. The routine ID matches the ROUTINEID column in SYSCAT.ROUTINES which can be used to retrieve the name of the calling procedure. If the executing routine is called 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 following 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.

The `COM.ibm.db2.app.UDF` class contains the following methods for facilitating the execution of Java UDFs in a partitioned database environment:

```
public int[] getDBPartitions() throws Exception
```

This function returns a list of all the partitions included in the table function.

```
public int getCurrentDBPartitionNum() throws Exception
```

This function returns the partition number of the node on which the table function is currently executing.

The `COM.ibm.db2.app.UDF` class contains the following methods for getting information required to create external generic table functions:

```
public int getNumColumns() throws Exception
```

For table UDFs, this function returns the number of output columns. For other UDFs, this function returns "1".

```
public int getColumnType(int position) throws Exception
```

This function returns the data type of the specified output column.

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.

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.

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 represented 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.

Passing parameters of data type ARRAY to Java routines

You can pass ARRAY data type parameters to and from Java procedures.

About this task

When your stored procedure is created with the ARRAY data type parameters, your stored procedure can accept or return a variable number of input data or return data of the same data type with a single parameter.

For example, you can pass all the names of students in a class to a procedure without knowing the number of students with a single parameter.

Procedure

To pass a parameter of type ARRAY:

1. The ARRAY data type must be already defined. To define an array type, the CREATE TYPE statement must be executed.
2. The procedure definition must include a parameter of the defined type. The following CREATE PROCEDURE statement example accepts a user-defined ARRAY data type `IntArray`:

Example

```
CREATE PROCEDURE inArray (IN input IntArray)
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'MyProcs:MyArrayProcs!inArray';
```

In the procedure definition, the array parameter is typed as java.sql.Array. Within the procedure, the argument is mapped to a Java array using the `getArray()` method, as shown in the following example. Notice the use of Integer rather than int (or other primitive types) for arrays.

```
static void inArray(java.sql.Array input)
{
Integer[] inputArr = (Integer [])input.getArray();
int sum = 0;
for(int i=0, i < inputArr.length; i++)
{
sum += inputArr[i];
}
}
```

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 - refer to the previously shown function signature example) to the value of the parameter marker in the query statement. A parameter marker is indicated with a "?" or a colon, followed by a name (:*name*).

```
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.

What to do next

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

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 -- refer to the previously shown function signature example) 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();
```

What to do next

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

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 "?" or by a colon followed by a name (:*name*).

```
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();
```

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();
  }
```

Restrictions on Java routines

There are multiple restrictions that apply to Java routines that you need to review before you create these routines.

The following restrictions apply to Java routines:

- The built-in procedure `install_jar` used to deploy Java routine code in JAR files to the database server file system cannot be called in a Java database application when you use the Db2Universal JDBC Driver. This driver does not support the `install_jar` procedure.

  The alternative is to use the Db2Command Line Processor.
- The PROGRAM TYPE MAIN clause is not supported in CREATE PROCEDURE or CREATE FUNCTION statements for Java routines regardless of the PARAMETER STYLE clause value specified.
- The following features are not supported by parameter style JAVA:
  - Table functions
  - Scratchpads in functions
  - Access to the DBINFO structure in functions
  - FINAL CALL invocation in functions

  The alternative is to create a Java function that use parameter style DB2GENERAL or to create the function in either the C or C++ programming language.
- Java Native Interface (JNI) calls from Java routines are not supported.

  If you need to invoke C or C++ code from a Java routine, you can do so by invoking a separately defined C or C++ routine.
- NOT FENCED Java routines are currently not supported. A Java routine that is defined as NOT FENCED is invoked as FENCED THREADSAFE.
- Java stored procedures cannot depend on any non-system resources, such as properties files. If you call a Java stored procedure that depends on non-system resources, those resources are not loaded, and no error is returned.
- Java generic table functions must be created using the DB2GENERAL parameter style.
- Modifications to the Java Virtual Machine (JVM) or the JVM start arguments is not supported for external Java routines.

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 during the processing of a statement by the database manager.

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.
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.

Note:

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.

Creating Java routines

Creating Java routines consists of executing a CREATE statement that defines the routine in a database server, and developing the routine implementation that corresponds to the routine definition.

Before you begin

- Review “Java routines” on page 163.
- Ensure that you have access to a database server, including instances and databases.
- Ensure that the operating system is at a version level that is supported by the database products.

- Ensure that the Java development software is at a version level that is supported for Java routine development. Refer to "Supported Java routine development software" on page 164.
- Ensure a valid "Specification of a driver for Java routines" on page 165 development.
- Authority to execute the CREATE PROCEDURE or CREATE FUNCTION statement.

For a list of restrictions associated with Java routines see:

- "Restrictions on Java routines" on page 183

About this task

The ways in which you can create Java routines follow:

- Using IBM Data Studio
- Using IBM IBM Data Studio
- Using the database routine development features in IBM Rational Application Developer
- Using the Db2 command window

In general it is easiest to create Java routines using the IBM Data Studio, although many developers enjoy the ability to create Java routines from within the integrated Java development environment provided by IBM Rational Application Developer. If these graphical tools are not available for use, the Db2 command window provides similar support through a command line interface.

Create Java routines using one of the following procedures:

Procedure

- Creating Java routines using IBM Data Studio
- Creating Java routines using Rational Application Developer
- "Creating Java routines from the command line" on page 185

Creating Java routines from the command line

Procedures and functions that reference a Java class are created in a similar way to external routines with other implementations. Steps for creating a Java routine include the formulation of the CREATE statement, coding, compilation (translation), and deployment of the Java class to the database server.

Before you begin

- Review "Java routines" on page 163.
- Ensure that you have access to the target database.
- Ensure that the operating system level is supported by the database product.
- Ensure that the Java development software is at a version level that is supported for Java routine development. Refer to "Supported Java routine development software" on page 164.
- Ensure a valid "Specification of a driver for Java routines" on page 165 development.
- Authority to execute the CREATE PROCEDURE or CREATE FUNCTION statement.
- The SQL procedures are implemented internally as packages, which have a different name. You can list package names that are associated with SQL procedures with the following query

About this task

You would choose to implement a Java routine 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 using Java and one of the JDBC or SQLJ application programming interfaces.

Procedure

1. Code the routine logic in Java.
   - A routine parameter signature must be implemented using one of the supported parameter styles. It is strongly recommended that `parameter style JAVA` be used for all Java routines. For more on parameter signatures and parameter implementations see:
     - “Parameters in Java routines” on page 169
     - “Parameter style JAVA procedures” on page 169
     - “Parameter style JAVA Java functions and methods” on page 170
   - Declare variables in the same manner as is done for Java database applications. Be careful to correctly use data types that map to SQL data types.

     For more on data type mapping between SQL and Java data types, see the following topics:
     - "Data types that map to database data types in Java applications" in *Developing Java Applications*
   - Include routine logic. Routine logic can consist of any code supported in the Java programming language. It can also include the execution of SQL statements in the same manner as is done in Java database applications.

     For more on executing SQL statements in Java code see:
     - "JDBC interfaces for executing SQL" in *Developing Java Applications*
     - "SQL statement execution in SQLJ applications" in *Developing Java Applications*
   - If the routine is a procedure and you might 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 Java routines:
     - “Returning result sets from JDBC procedures” on page 179
     - “Returning result sets from SQLJ procedures” on page 180
   - Set a routine return value at the end of the routine.
2. Build your code to produce a Java class file or JAR file containing a collection of Java class files.

   For information on how to build Java routine code, see:
   - "Building JDBC routines" in *Developing Java Applications*
   - "Building SQL routines" in *Developing Java Applications*
3. Copy the class file to the database server or install the JAR file to the database server.

   For information on how to do this, see:
   - “Deploying Java routine class files to database servers” on page 190
   - “JAR file administration on the database server” on page 191

   It is recommended that you store class files associated with database routines in the *function directory*. To find out more about the function directory, see information related to the EXTERNAL clause in one of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the library 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 library as you will require it for the next step.
4. Execute either dynamically or statically the appropriate CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION.
   - Specify the LANGUAGE clause with: JAVA
   - Specify the PARAMETER STYLE clause with the name of the supported parameter style that was implemented in the routine code. It is strongly recommended that PARAMETER STYLE JAVA be

used unless the features you require are only supported when PARAMETER STYLE DB2GENERAL is used.

- Specify the EXTERNAL clause with the name of the JAR file or Java class to be associated with the routine using one of the following values:
  - the fully qualified path name of the Java class file
  - the relative path name of the routine Java class file relative to the function directory.
  - the JAR file ID of the JAR file on the database server that contains the Java class

  By default the database manager searches for the library in the function directory unless a JAR file ID and class, fully qualified path name, or relative path name for it is specified in the EXTERNAL clause.
- Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure and it will return one or more result sets to the caller.
- Specify any other non-default clause values in the CREATE statement to be used to characterize the routine.

What to do next
To invoke your Java routine, see "Invoking routines" on page 205.

Building Java routine code
Once Java routine implementation code is written, it must be compiled and deployed before the routine can be called. The steps that are required to build Java routines are similar to steps for building other external routine. However, there are some differences.

Procedure

You can use one of the following tools to build Java routines:

- The graphical tool that is provided with IBM Data Studio
- The graphical tool that is provided within IBM IBM Data Studio
- The graphical tool that is provided within IBM Rational Application Developer
- The database sample build-scripts
- The Db2 command window

The sample build scripts and batch files for routines are designed for building sample routines (procedures and user-defined functions) in a particular operating system with the default supported development software.

There is a separate set of sample build scripts and batch files for Java routines that are created with JDBC and SQLJ. It is easier to build Java routines with the graphical tools or the build scripts, which can be modified as required. However, it is often helpful to know how to build routines from the Db2 command window.

Building JDBC routines
You can use a Java `makefile` or the javac command to build JDBC routines. After you build those routines, you need to catalog them.

About this task
The following steps demonstrate how to build and run these routines:

- The `SpServer` sample JDBC stored procedure
- The `UDFsrv` sample user-defined function, which has no SQL statements
- The `UDFsqlsv` sample user-defined function, which has SQL statements

Procedure

- To build and run the SpServer.java stored procedure on the server, from the command line:
  a) Compile SpServer.java to produce the file SpServer.class with this command:

```
javac SpServer.java
```

  b) Copy the `SpServer.class` file to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on Linux or UNIX operating systems.
  c) Catalog the routines by running the spcat script on the server.

  The spcat script connects to the sample database, uncatalogs the routines if they were previously cataloged by calling SpDrop.db2, then catalogs them by calling SpCreate.db2, and finally disconnects from the database. You can also run the SpDrop.db2 and SpCreate.db2 scripts individually.
  d) Stop and restart the database to allow the new class file to be recognized. If necessary, set the file mode for the class file to "read" so it is readable by the fenced user.
  e) Compile and run the `SpClient` client application to access the stored procedure class.
- To build and run the `UDFsrv.java` user-defined function program (user-defined function with no SQL statements) on the server, from the command line:
  a) Compile `UDFsrv.java` to produce the file UDFsrv.class with this command:

```
javac UDFsrv.java
```

  b) Copy `UDFsrv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on Linux and UNIX operating systems.
  c) Compile and run a client program that calls UDFsrv.

  To access the `UDFsrv` library, you can use the UDFcli.java JDBC application, or the UDFcli.sqlj SQLJ client application. Both versions of the client program contain the CREATE FUNCTION SQL statement that you use to register the user-defined functions with the database, and also contain SQL statements that use the user-defined functions.
- To build and run the `UDFsqlsv.java` user-defined function program (user-defined function with SQL statements) on the server, from the command line:
  a) Compile `UDFsqlsv.java` to produce the file `UDFsqlsv.class` with this command:

```
javac UDFsqlsv.java
```

  b) Copy `UDFsqlsv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on Linux and UNIX operating systems.
  c) Compile and run a client program that calls UDFsqlsv.

  To access the `UDFsqlsv` library, you can use the `UDFsqlcl.java` JDBC application. The client program contains the CREATE FUNCTION SQL statement that you use to register the user-defined functions with the database, and also contains SQL statements that use the user-defined functions.

Building SQLJ routines

You can use a Java `makefile` or the `bldsqljs` build file to build SQLJ routines. After you build those routines, you need to catalog them.

About this task

The following steps demonstrate how to build and run the `SpServer` sample SQLJ stored procedure. These steps use the build file, `bldsqljs` (Linux and UNIX), or `bldsqljs.bat` (Windows), which contains commands to build either an SQLJ applet or application.

The build file takes up to six parameters: $1, $2, $3, $4, $5, and $6 on Linux and UNIX operating systems, and %1, %2, %3, %4, %5, and %6 on Windows operating systems. The first parameter specifies the name of your program. The second parameter specifies the user ID for the database instance. The third parameter specifies the password. The fourth parameter specifies the server name. The fifth

parameter specifies the port number. And the sixth parameter specifies the database name. For all but the first parameter, program name, default values can be used. See the build file for details about using default parameter values.

Procedure

1. Build the stored procedure application with this command:

   ```
   bldsqljs SpServer <userid> <password> <server_name> <port_number> <db_name>
   ```

2. Catalog the stored procedure with this command:

   ```
   spcat
   ```

   This script connects to the sample database, uncatalogs the routines if they were previously cataloged by calling `SpDrop.db2`, then catalogs them by calling `SpCreate.db2`, and finally disconnects from the database. You can also run the `SpDrop.db2` and `SpCreate.db2` scripts individually.
3. Stop and restart the database to allow the new class file to be recognized. If necessary, set the file mode for the class file to read, so it is readable by the fenced user.
4. Compile and run the `SpClient` client application to access the stored procedure class.

   You can build `SpClient` with the application build file, `bldsqlj` (Linux and UNIX), or `bldsqlj.bat` (Windows).

Compile and link options for Java (SQLJ) routines

SQLJ routine options for Linux and UNIX
The `bldsqljs` build script builds SQLJ routines on the Linux and UNIX operating systems. `bldsqljs` specifies a set of SQLJ translator and customizer options.

Recommendation: Use the same SQLJ translator and customizer options that `bldsqljs` uses when you build your SQLJ routines on the Linux and UNIX platforms.

The options that `bldsqljs` includes are:

sqlj
The SQLJ translator (also compiles the program).

"${progname}.sqlj"
The SQLJ source file. The `progname=${1%.sqlj}` command removes the extension if it was included in the input file name, so when the extension is added back again, it is not duplicated.

db2sqljcustomize
The SQLJ profile customizer.

-url
Specifies a JDBC URL for establishing a database connection, such as `jdbc:db2://servername:50000/sample`.

-user
Specifies a user ID.

-password
Specifies a password.

"${progname}_SJProfile0"
Specifies a serialized profile for the program.

SQLJ routine options for Windows
The `bldsqljs.bat` batch file builds SQLJ routines on Windows operating systems. `bldsqljs.bat` specifies a set of SQLJ translator and customizer options.

You can use the same SQLJ translator and customizer options that are specified in the `bldsqljs.bat` batch file when you build your SQLJ routines on Windows operating systems.

The following SQLJ translator and customizer options are used in the `bldsqljs.bat` batch file on Windows operating systems. You can use these options to build SQLJ routines (stored procedures and user-defined functions).

sqlj
The SQLJ translator (also compiles the program).

%1.sqlj
The SQLJ source file.

db2sqljcustomize
The SQLJ profile customizer.

-url
Specifies a JDBC URL for establishing a database connection, such as `jdbc:db2://servername:50000/sample`.

-user
Specifies a user ID.

-password
Specifies a password.

%1_SJProfile0
Specifies a serialized profile for the program.

Deploying Java routine class files to database servers
Java routine implementations must be deployed to the database server file system so that they can be located, loaded, and run upon routine invocation.

One or more Java routine implementations can be included in an individual Java class file. Java class files that contain Java routine implementations can be collected together into JAR files. The Java class files that you use to implement a routine must reside in a JAR file that is installed in the target database.

Prerequisites:

- Identify the database server to which you want to deploy the routine class.
- On UNIX and Linux operating systems, identify the user ID of the database instance owner. If this is not known, see your database administrator.
- Determine if the location where you deploy the routine class is accessible on all of the hosts in the `db2nodes.cfg` file. Consider packaging your routine class files into a JAR file and installing the JAR on the database server with the SQLJ.INSTALL_JAR procedure.

To deploy individual Java routine class files:

- Copy the Java routine class into the database function directory.

  On UNIX and Linux operating systems, the function directory is defined as: *install_path*`/function` where *install_path* is the database manager installation path. For example, *$HOME*`/sqllib/function`, where *$HOME* is the home directory of the instance owner.

  On Windows operating systems, the function directory is defined as: *instance_profile_path*`\function` where the *instance_profile_path* is the path, which was specified in the **db2icrt** (create instance) command. You can find the instance profile path name by issuing the **db2set** command as follows:

  ```
  db2set DB2INSTPROF
  ```

  For example, `C:\Documents and Settings\All Users\Application Data\IBM\DB2\db2copy1\function`

  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 called ibm.tests.test1 for a Linux operating system, store the corresponding Java bytecode file (named `test1.class`) in *$HOME*`/sqllib/function/ibm/tests`, where *$HOME* is the home directory of the instance owner.

To deploy JAR files that contain Java routine class files:

- You must install the JAR file that contains Java routine class files to the database server file system. Refer to "JAR file administration on the database server" on page 191.

Once the Java routine class files is deployed and the CREATE statement has been executed to define the routine in the database, you can invoke the routine.

Deploying Java routine class files to database servers with dependent classes

When the Java routine class files have dependencies on classes that are not part of the standard Java or database classes, repeat the steps that are identified in the previous section for each dependent class.

Alternatively, the database can be configured to search the directories in the **CLASSPATH** environment variable in order to detect dependant classes. On Windows operating systems, the database manager searches the specified directories in the **CLASSPATH** system environment variable. On UNIX and Linux operating systems, the database manager searches the instance owner's **CLASSPATH** environment variable if the text " CLASSPATH " is specified as part of the **DB2ENVLIST** environment variable. It is strongly recommended that dependant classes be installed rather than relying on the **CLASSPATH** environment variable.

JAR file administration on the database server

To deploy JAR files that contain Java routine class files, you must install the JAR file to the database server. This can be done from an IBM Data Server Client by using built-in routines that install, replace, or remove JAR files on the database server.

To install, replace, or remove a JAR file in a database instance, use the stored procedures that are provided with the database product:

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
- DBADM authority

```
sqlj.db2_install_jar( jar-locator, jar-id )
```

Replace

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

```
sqlj.db2_replace_jar( jar-locator, 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.
- *jar-locator*: A BLOB locator input parameter that points to the JAR file that is to be installed in the Db2 catalog.

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, the database stores JAR files in the path that is 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.

Note: On a partitioned database system, the command `sqlj.install_jar()` only installs the JAR file on the node issuing the command. Run `sqlj.replace_jar()` on each node to copy the *jar* file.

Updating Java routine classes

If you want to change the logic of a Java routine, you must update the routine source code, compile (translate) the code, and then update the version of the Java class or JAR file that is deployed to the database server.

About this task

To ensure that the database manager uses the new version of the Java routine, you must execute a built-in procedure that loads the new version of the Java class into memory.

Procedure

To update Java routine classes:

1. Deploy the new Java class or JAR file to the database server.
2. Execute the following built-in procedure for fenced routines:

   ```
   CALL SQLJ.REFRESH_CLASSES()
   ```

   The **CALL SQLJ.REFRESH_CLASSES()** statement forces the database manager to load the new class into memory upon the next commit or rollback operation.

   The **CALL SQLJ.REFRESH_CLASSES()** statement does not affect the unfenced routines. For unfenced routines, you must explicitly stop and restart the database manager in order for new versions of Java routine classes to be loaded and used.

   Note: On a partitioned database system, loading new classes requires you to issue the `sqlj.refresh_classes()` command on all database partitions. This can be done with the `db2_all` command. In order for this command to work, you need to connect to the database first, and use \" around the call command. Otherwise a syntax error will appear. This is shown by this example:

   ```
   db2_all "db2 connect to sample; db2 \"call sqlj.refresh_classes();\""
   ```

Results

If you do not perform the steps listed previously, after you update Java routine classes, the database manager will continue to use the previous versions of the classes.

Examples of Java (JDBC) routines

When developing Java routines that use the JDBC application programming interface, it is helpful to refer to examples to get a sense of what the CREATE statement and the Java routine code should look like.

About this task

The following topics contain examples of Java procedures and functions:

Procedure

- Examples of Java (JDBC) procedures
- Examples of Java (JDBC) procedures with XML features
- Examples of Java (JDBC) functions

Example: Array data type in Java (JDBC) procedure

An example of a Java routine using the array data type.

The following example illustrates the skeleton of a Java routine with an IN and an OUT parameter of the array data type.

```
CREATE TYPE phonenumbers AS VARCHAR(20) ARRAY[10] %
CREATE PROCEDURE javaproc( IN in1 phonenumbers,
                           OUT out1 phonenumbers)
  LANGUAGE java
  PARAMETER STYLE java
  FENCED THREADSAFE
  EXTERNAL NAME 'myjar:stpclass.javastp' %
```

```
import java.sql.Array;

public static void javaproc(Array input, Array[] output)
{
output[0] = input;
}
```

Example: XML and XQuery support in Java (JDBC) procedure

Once the basics of Java procedures, programming in Java using the JDBC application programming interface (API), and XQuery are understood, you can create and call Java procedures that query XML data.

This example of a Java procedure illustrates:

- the **CREATE PROCEDURE** statement for a parameter style JAVA procedure
- the source code for a parameter style JAVA procedure
- input and output parameters of data type XML
- use of an XML input parameter in a query
- assignment of the result of an XQuery, an XML value, to an output parameter
- assignment of the result of an SQL statement, an XML value, to an output parameter

Prerequisites

Before you work with the Java procedure example, you might want to read the following topics:

- Java routines
- Routines
- Building Java routine code

The examples use a table that is named `xmlDataTable`, which is created and populated with the following statements:

```
CREATE TABLE xmlDataTable
(
   num INTEGER,
   xdata XML
)@
```

```
  INSERT INTO xmlDataTable VALUES
     (1, XMLPARSE(DOCUMENT '<doc>
                               <type>car</type>
                               <make>Pontiac</make>
                               <model>Sunfire</model>
                               </doc>' PRESERVE WHITESPACE)),
     (2, XMLPARSE(DOCUMENT '<doc>
                               <type>car</type>
                               <make>Mazda</make>
                               <model>Miata</model>
                               </doc>' PRESERVE WHITESPACE)),
     (3, XMLPARSE(DOCUMENT '<doc>
                               <type>person</type>
                               <name>Mary</name>
                               <town>Vancouver</town>
                               <street>Waterside</street>
                               </doc>' PRESERVE WHITESPACE)),
     (4, XMLPARSE(DOCUMENT '<doc>
                               <type>person</type>
                               <name>Mark</name>
                               <town>Edmonton</town>
                               <street>Oak</street>
                               </doc>' PRESERVE WHITESPACE)),
     (5, XMLPARSE(DOCUMENT '<doc>
                               <type>animal</type>
                               <name>dog</name>
                               </doc>' PRESERVE WHITESPACE)),
     (6, NULL),
     (7, XMLPARSE(DOCUMENT '<doc>
                               <type>car</type>
                               <make>Ford</make>
                               <model>Taurus</model>
                               </doc>' PRESERVE WHITESPACE)),
     (8, XMLPARSE(DOCUMENT '<doc>
                               <type>person</type>
                               <name>Kim</name>
                               <town>Toronto</town>
                               <street>Elm</street>
                               </doc>' PRESERVE WHITESPACE)),
     (9, XMLPARSE(DOCUMENT '<doc>
                               <type>person</type>
                               <name>Bob</name>
                               <town>Toronto</town>
                               <street>Oak</street>
                               </doc>' PRESERVE WHITESPACE)),
     (10, XMLPARSE(DOCUMENT '<doc>
                                <type>animal</type>
                                <name>bird</name>
                                </doc>' PRESERVE WHITESPACE))@
```

Procedure

You can use the following example as references when you are making your own Java procedures:

- “The Java external code file” on page 194
- “Example: Parameter style JAVA procedure with XML parameters” on page 195

The Java external code file

The example shows a Java procedure implementation. The example consists of two parts: the **CREATE PROCEDURE** statement and the external Java code implementation of the procedure from which the associated Java class can be built.

The Java source file that contains the procedure implementations of the following examples is named `stpclass.java` included in a JAR file named `myJAR`. The file has the following format:

```
using System;
import java.lang.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import com.ibm.db2.jcc.DB2Xml;

 public class stpclass
 {          ...
    // Java procedure implementations
```

```
            ...
 }
```

The Java class file imports are indicated at the top of the file. The `com.ibm.db2.jcc.DB2Xml` import is required if any of the procedures in the file contain parameters or variables of type XML is used.

It is important to note the name of the class file and JAR name that contains the procedure implementation. These names are important because the EXTERNAL clause of the **CREATE PROCEDURE** statement for each procedure must specify this information so that the database manager can locate the class at run time.

Example: Parameter style JAVA procedure with XML parameters

The following example contains the Java code for a parameter style JAVA procedure with XML parameter and corresponding **CREATE PROCEDURE** statement. The `xmlProc1` procedure takes an input parameter, `inXML`, inserts the `inXML` value into a table, queries XML data using both an SQL statement and an XQuery expression, and sets two output parameters, `outXML1`, and `outXML2`.

```
CREATE PROCEDURE  xmlProc1   ( IN inNUM  INTEGER,
                               IN inXML  XML as CLOB (1K),
                               OUT out1XML XML as CLOB (1K),
                               OUT out2XML XML as CLOB (1K)
                                  )
DYNAMIC RESULT SETS 0
DETERMINISTIC
LANGUAGE JAVA
PARAMETER STYLE JAVA
MODIFIES SQL DATA
FENCED
THREADSAFE
DYNAMIC RESULT SETS 0
PROGRAM TYPE SUB
NO DBINFO
EXTERNAL NAME 'myJar:stpclass.xmlProc1'@
```

```
//*************************************************************************
//  Stored Procedure: XMLPROC1
//
//  Purpose:   Inserts XML data into XML column; queries and returns XML data
//
//  Parameters:
//
//   IN:      inNum -- the sequence of XML data to be insert in xmldata table
//            inXML -- XML data to be inserted
//   OUT:     out1XML -- XML data to be returned
//            out2XML -- XML data to be returned
//
//*************************************************************************
```

```
public void xmlProc1(int inNum,
                     DB2Xml inXML ,
                     DB2Xml[] out1XML,
                     DB2Xml[] out2XML
                    )
throws Exception
{
       Connection con = DriverManager.getConnection("jdbc:default:connection");

       // Insert data including the XML parameter value into a table
       String query = "INSERT INTO xmlDataTable (num, inXML ) VALUES ( ?, ? )" ;
       String xmlString = inXML.getDB2String() ;

       stmt = con.prepareStatement(query);
       stmt.setInt(1, inNum);
       stmt.setString (2, xmlString );
       stmt.executeUpdate();
       stmt.close();

       // Query and retrieve a single XML value from a table using SQL
       query = "SELECT xdata from xmlDataTable WHERE num = ? " ;

       stmt = con.prepareStatement(query);
       stmt.setInt(1, inNum);
       ResultSet rs = stmt.executeQuery();
```

```
        if ( rs.next() )
        {  out1Xml[0] = (DB2Xml) rs.getObject(1);  }

        rs.close() ;
        stmt.close();

        // Query and retrieve a single XML value from a table using XQuery
        query = "XQUERY for $x in db2-fn:xmlcolumn(\"xmlDataTable.xdata\")/doc
                                where $x/make = \'Mazda\'
                                return <carInfo>{$x/make}{$x/model}</carInfo>";

        stmt = con.createStatement();

        rs = stmt.executeQuery( query );

        if ( rs.next() )
        {  out2Xml[0] = (DB2Xml) rs.getObject(1) ;  }

        rs.close();
        stmt.close();
        con.close();

        return ;
}
```

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, the database manager act as an OLE automation controller. Through this mechanism, the database manager 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.

Creating OLE automation routines

OLE automation routines are implemented as public methods of OLE automation objects.

About this task

The OLE automation objects must be creatable by an OLE automation controller out side of the database 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, because 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)
```

OLE routine object instances and scratchpad considerations

OLE automation UDFs and methods of OLE automation objects are applied on instances of OLE automation objects. The database manager 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.

Supported SQL data types in OLE automation

The database manager handles type conversion between SQL types and OLE automation types.

The following table summarizes the supported data types and how they are mapped.

Table 12. 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

Table 12. Mapping of SQL and OLE Automation Datatypes (continued)		
SQL Type	**OLE Automation Type**	**OLE Automation Type Description**
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	64-bit floating-point fractional number of days since December 30, 1899
TIMESTAMP	DATE	64-bit floating-point fractional number of days since December 30, 1899
CHAR(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARCHAR(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
LONG VARCHAR	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
CLOB(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
GRAPHIC(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARGRAPHIC(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
LONG GRAPHIC	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
DBCLOB(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
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]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)

Table 12. Mapping of SQL and OLE Automation Datatypes (continued)		
SQL Type	**OLE Automation Type**	**OLE Automation Type Description**
LONG VARCHAR	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)
CHAR(*n*) FOR BIT DATA	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*) FOR BIT DATA	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)
BLOB(*n*)	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)

Data that is passed between the database and OLE automation routines is passed as call by reference. The following types are not supported:

- SQL types such as BIGINT, DECIMAL, BINARY, VARBINARY, or LOCATORS
- OLE automation types such as Boolean or CURRENCY

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).

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 13. 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

Table 13. Mapping of SQL and OLE data types to BASIC and C++ data types (continued)			
SQL Type	**OLE Automation Type**	**BASIC Type**	**C++ Type**
FLOAT or DOUBLE	double	Double	double
DATE, TIME, TIMESTAMP	DATE	Date	DATE
CHAR(*n*)	BSTR	String	BSTR
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
VARCHAR(*n*)	BSTR	String	BSTR
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
LONG VARCHAR	BSTR	String	BSTR
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
BLOB(*n*)	BSTR	String	BSTR
BLOB(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
GRAPHIC(*n*), VARGRAPHIC(*n*), LONG GRAPHIC, DBCLOB(*n*)	BSTR	String	BSTR

OLE Automation in BASIC

To implement OLE automation routines in BASIC, you need to use the BASIC data types corresponding to the SQL data types mapped to OLE automation types.

The following declaration example is for the OLE automation UDF `bcounter`, which is created with the BASIC programming language:

```
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 following declaration example is for the OLE automation UDF `increment`, which is created with the C ++ programming language:

```
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 previously shown 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);
```

You can use the ODL description 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 routine assigns the parameter a new value. The same rule applies for one-dimensional byte arrays that are received by the called routine as SAFEARRAY**. The following list contains OLE imposed rules on parameters:

- [in] parameters: The database manager allocates and frees [in] parameters.
- [out] parameters: The database manager passes in a pointer to NULL. The [out] parameter must be allocated by the routine that is called and is freed by the database manager.
- [in,out] parameters: The database manager initially allocates [in,out] parameters. They can be freed and reallocated by the routine that is called. As is true for [out] parameters, the database manager frees the final returned parameter.

All other parameters are passed as pointers. The database manager allocates and manages the referenced memory.

OLE automation provides a set of data manipulation functions for dealing with BSTRs and SAFEARRAYs.

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.
     // Database manager 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*. When an OLE automation server is implemented as *creatable single-use*, each client (that is, a FENCED process) that connects with the CoGetClassObject function to an OLE automation object uses its own instance of a class factory, and run a new copy of the OLE automation server. When an OLE automation server is implemented as *creatable multi-use*, many clients connect to the same class factory. In an OLE automation server that is implemented as *creatable multi-use*, each instantiation of a class factory is supplied by an already running copy of the OLE server. 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 can provide better performance.

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.

The database manager simplifies the creation of OLE DB applications by enabling you to define table functions that access an OLE DB data source. The database manager 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 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 the 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, the database returns 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 database.
- BINARY and VARBINARY data types are not supported as parameters or return values.

Creating an OLE DB table UDF

Create an OLE DB table function by using a single CREATE FUNCTION statement. You can use OLE DB table functions to provide built-in access to any OLE DB provider, which reduces the amount of effort that is required for application development.

Procedure

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. The database passes the command text to the OLE DB provider for execution and the OLE DB provider returns a rowset to the database. Column names and data types of the resulting rowset need to be compatible with the RETURNS TABLE definition in the CREATE FUNCTION statement. You must ensure that you name the columns properly, because binding of the column names of the rowset is based on matching column names.

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. The table function has an input parameter to pass through command text to the OLE DB provider, so 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;
```

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.

Supported SQL data types in OLE DB
The database data types map to the OLE DB data types.

The following table shows how the database 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 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 14. Mapping of the database data types to the OLE DB data types

Database Data Type	OLE DB Data Type
SMALLINT	DBTYPE_I2

Table 14. Mapping of the database data types to the OLE DB data types (continued)	
Database Data Type	**OLE DB Data Type**
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

Note: OLE DB data type conversion rules are defined in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. For example:

- To retrieve the OLE DB data type DBTYPE_CY, the data can get converted to OLE DB data type DBTYPE_NUMERIC(19,4), which maps to the database 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 the database 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 the database data type CHAR(12) FOR BIT DATA.

Invoking routines

Once a routine has been developed and created in the database by issuing the CREATE statement, if the appropriate routine privileges 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 might 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 212
- “Calling procedures from triggers or SQL routines” on page 213
- See "Call a procedure from a CLI application" in *Call Level Interface Guide and Reference Volume 1*
- “Calling procedures from the command line processor (CLP)” on page 215
- “Invoking scalar functions or methods” on page 220
- “Invoking user-defined table functions” on page 220

Authorizations and binding of routines that contain SQL

To successfully invoke routines, you must have multiple authorizations and bindings 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** or **BIND** command parameter 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
- DATAACCESS 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 any user with ACCESSCTRL or SECADM authority, CONTROL or EXECUTE WITH GRANT OPTION privilege on the package. (The creator of a package automatically receives CONTROL and EXECUTE WITH GRANT OPTION on the package.)

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** or **BIND** command parameter 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), ACCESSCTRL, or SECADM 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** or **BIND** command parameter. In this case, the invoker of the routine assumes this value rather than the ID of the user who is executing the application.

Note:

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:

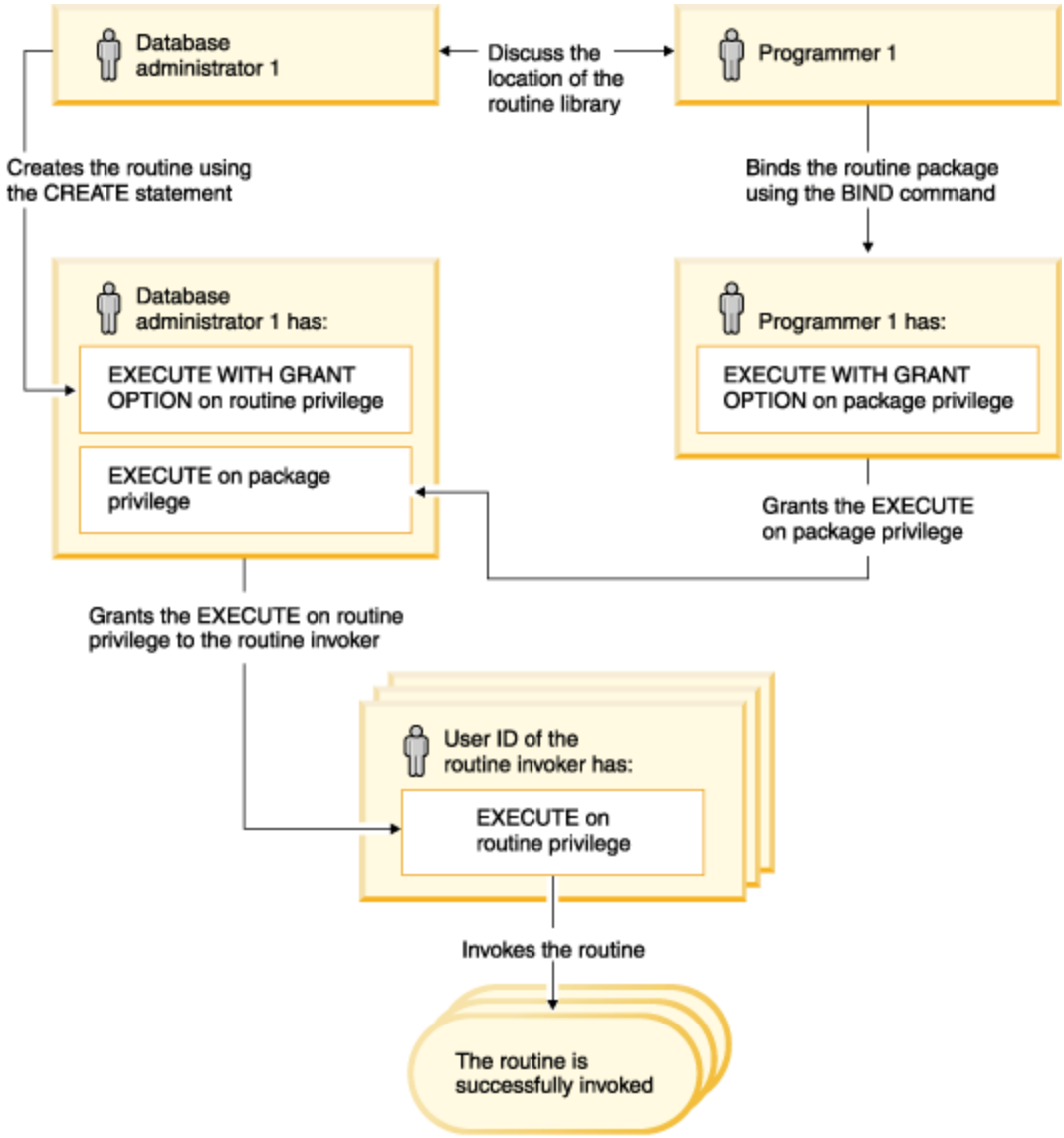


Figure 3. Managing the EXECUTE privilege on routines

1. Definer performs the appropriate CREATE statement to register the routine. This registers the routine in the database 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, the database manager 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).

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. The qualified name of a method is `schema-name.type.method-name`.

For example:

```
SANDRA.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
```

However, you can also omit the `schema-name.`, in which case, the database manager will attempt to identify the stored procedure or UDF to which you are referring. For example:

```
BOAT_COMPARE    FOO    SUBSTR    FLOOR
```

The concept of *SQL path* is central to the 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 re-implement 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.

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 routines under the following restrictions:

64 levels of nesting
You can nest routine invocations up to 64 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 61 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.

Invoking 32-bit routines on a 64-bit database server

In 64-bit database instances, it is possible to invoke C and COBOL routines that reference 32-bit external routine libraries, however these routines must be specified to run as fenced and not threadsafe.

About this task

This is done by including both the FENCED clause and NOT THREADSAFE clause in the routine CREATE statement when creating a new routine. For routines that have already been created in the 64-bit instance, the ALTER FUNCTION or ALTER PROCEDURE statements can be used to modify the routine definition. The first time such a 32- bit routine is invoked in a 64- environment, there will be a performance degradation. Subsequent invocations of the 32-bit stored procedure will perform as well as an equivalent 64-bit routine. Use of 32-bit routines in 64-bit database instances is discouraged.

To successfully invoke Java procedures in a 64-bit database instance on a 64-bit database server, a 64-bit Java Virtual Machine (JVM) is required. 32-bit JVMs are not supported for running routines in 64-bit database instances. As Java classes are platform independent, a Java class compiled with a 32-bit software development kit can run successfully with a 64-bit JVM. Routine performance is not impacted by doing this.

Procedure

To invoke existing 32-bit routines on a 64-bit server:

1. Copy the routine class or library to the database routines directory:
   - Linux and 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.

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. The database manager does not attempt to shuffle arguments to better match a stored procedure definition, and the database manager do not understand the semantics of the individual stored procedure parameters.

Calling procedures

Once the activities required to create a procedure (also called a stored procedure) have been completed, a procedure can be invoked by using the CALL statement. The CALL statement is an SQL statement that enables the procedure invocation, the passing of parameters to the procedure, and the receiving of parameters returned from the procedure.

About this task

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:

- an embedded SQL client application
- an external routine (procedure, UDF, or method)
- an SQL routine (procedure, UDF, or method)
- an SQL trigger (BEFORE TRIGGER, AFTER TRIGGER, or INSTEAD OF TRIGGER)
- an SQL dynamic compound statement
- from the 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 platform 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.

An autonomous procedure is a procedure that, when called, executes inside a new transaction independent of the original transaction. When the autonomous procedure successfully completes, it will commit the work performed within the procedure, but if it is unsuccessful, the procedure rolls back any work it performed. Whatever the result of the autonomic procedure, the transaction which called the autonomic procedure is unaffected. To specify a procedure as autonomous, specify the AUTONOMOUS keyword on the CREATE PROCEDURE statement.

When you call a procedure, certain rules apply about exactly which procedure is selected. Procedure selection depends partly on whether you qualify the procedure by specifying the schema name. The database manager also performs checks based on the number of arguments and any argument names specified in the call to the procedure. See information about the CALL statement for more details about procedure selection.

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.

Before you begin

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.

Results

Note: You can code SQL statements and routine logic at any point between steps 2 and 5.

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.

Before you begin

- 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 roll back 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 roll back any external actions effected by the procedures, such as writing to the file system.
- 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 from inline SQL PL statements.
- If a cursor defined as **WITH RETURN TO CLIENT** is opened during the execution of a compiled trigger, result sets from the cursor will be discarded.

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 routines” on page 38
- “SQL statements that can be executed in routines and triggers” on page 33

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) 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.

Results

Runtime errors might occur 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 following SQL statements create the necessary 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 or replace PROCEDURE proc(IN val INT, IN name CHAR(2))
LANGUAGE SQL
DYNAMIC RESULT SETS 0
MODIFIES SQL DATA
BEGIN
  DECLARE rc INT DEFAULT 0;
  INSERT INTO T2 VALUES (val, name);
  GET DIAGNOSTICS rc = ROW_COUNT;
  IF ( rc > 0 ) THEN
      RETURN 0;
  ELSE
      RETURN -200;
  END IF;
END@

CREATE or replace TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.c1 > 100)
BEGIN ATOMIC
   DECLARE rc INTEGER DEFAULT 0;
   CALL proc(n.c1, n.c2);
   GET DIAGNOSTICS rc = RETURN_STATUS;
   VALUES(CASE WHEN rc < 0 THEN CAST(RAISE_ERROR('70001', 'PROC CALL failed')
      as varchar(70))END);
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'@
```

Calling procedures from the command line processor (CLP)
You can call stored procedures by using the CALL statement from the command line processor interface. The stored procedures must be defined in the database system catalog tables.

Procedure

To call a 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 and a place-holder ('?') for each OUT parameter values.

The parameters for a stored procedure are defined in the **CREATE PROCEDURE** statement for the stored procedure.

Note: CLP call statement truncates the output parameter values to 8k for `blob`, `clob`, `dbclob`, `graphic`, `vargraphic`, and `longvargraphic` types.

Example

SQL procedure examples

Example 1.
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)
```

The `DEPT_MEDIAN` procedure selects the STAFF table for the specified `deptNumber` value. You can call the `DEPT_MEDIAN` procedure with the following CALL statement:

```
db2 call dept_median (51, ?)
```

On Linux and UNIX operating systems, the parentheses have special meaning to the command shell, so they must be preceded with a "\" character or surrounded with quotation marks, as follows:

```
db2 "call dept_median (51, ?)"
```

You do not use quotation marks if you are using the interactive mode of the command line processor. The following results are returned from the `DEPT_MEDIAN` procedure:

```
Value of output parameters
--------------------------
Parameter Name  : MEDIANSALARY
Parameter Value : +1.76545000000000E+004

Return Status = 0
```

Example 2.
The example 2 illustrates how to call a procedure with array parameters. The user-defined data type `phonenumbers` is defined as follows:

```
CREATE TYPE phonenumbers AS VARCHAR(12) ARRAY[1000]
```

In the following example, the `find_customers` procedure contains parameters of type `phonenumbers`. The `find_customers` procedure searches for the `area_code` value in the `numbers_in` parameter and reports them in the `numbers_out` parameter.

```
CREATE PROCEDURE find_customers(
IN numbers_in phonenumbers,
```

```
IN area_code CHAR(3),
OUT numbers_out phonenumbers)
BEGIN
DECLARE i, j, max INTEGER;

SET i = 1;
SET j = 1;
SET numbers_out = NULL;
SET max = CARDINALITY(numbers_in);

WHILE i <= max DO
IF substr(numbers_in[i], 1, 3) = area_code THEN
SET numbers_out[j] = numbers_in[i];
SET j = j + 1;
END IF;
SET i = i + 1;
END WHILE;
END
```

To call the `find_customers` procedure, you can use the following CALL statement:

```
db2 CALL find_customers(ARRAY['416-305-3745',
                              '905-414-4565',
                              '416-305-3746'],
                              '416',
                               ?)
```

As shown in the CALL statement example, when a procedure has an input parameter of an array data type, the input argument can be specified with an array constructor that contains a list of literal values.

The following results are returned from the `find_customers` procedure:

```
Value of output parameters
  --------------------------
  Parameter Name  : OUT_PHONENUMBERS
  Parameter Value : ['416-305-3745',
                     '416-305-3746']

  Return Status = 0
```

C stored procedure example

You can also call stored procedures that are created from supported host languages with the Command Line Processor. In the c subdirectory under the `sample` directory contains files that contain sample stored procedures. The `spserver` shared library contains a number of stored procedures that can be created from the `spserver.sqc` file. The `spcreate.db2` file registers the stored procedures.

In the `spcreate.db2` file, the following `CREATE PROCEDURE` statement for the `MAIN_EXAMPLE` procedure can be found:

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
                               OUT salary DOUBLE,
                               OUT errorcode INTEGER)
```

The `MAIN_EXAMPLE` procedure selects the `job` value from the EMPLOYEE table. The C sample program, `spclient`, that calls the stored procedure, uses `'DESIGNER'` for the `JOB` value. :

```
   db2 "call MAIN_EXAMPLE ('DESIGNER', ?, ?)"
```

The following results are returned from the `MAIN_EXAMPLE` procedure:

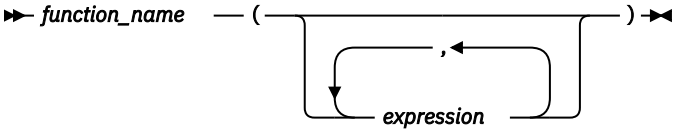
```
   Value of output parameters
   --------------------------
   Parameter Name  : SALARY
   Parameter Value : +2.37312500000000E+004

   Parameter Name  : ERRORCODE
   Parameter Value : 0

   Return Status = 0
```

An ERRORCODE of zero indicates a successful completion.

References to functions

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.

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 database manager does not attempt to shuffle arguments to better match a function definition, and do 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.

You cannot use the following code to specify the parameter markers in the BL0OP function:

```
BLOOP(?)
```

Because the function selection logic does not know what data types 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 these 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
```

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; and 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.

Exception: If there is an unqualified reference to a function named RID, and the function is invoked with a single argument that matches a table-reference in the FROM clause of the subselect, the schema is SYSIBM and the built-in RID function is invoked.

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
```

In this statement, if column1 is a DECIMAL or DOUBLE column, the inner BLOOP reference resolves to the second BLOOP. Because this BLOOP returns an INTEGER, the outer BLOOP resolves to the first BLOOP.

Alternatively in this DML statement, if column1 is a SMALLINT or INTEGER column, the inner bloop reference resolves to the first BLOOP. 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.

Distinct types as UDF or method parameters

The user-defined functions (UDFs) and methods can be defined with parameters that are distinct types or return distinct types result.

Procedure

You must explicitly cast distinct type arguments that originate from host variables when they are used to call an UDFs or a method that accepts distinct types parameters.

The explicit casting is required by the database, which uses strong typing and to avoid ambiguous results. The following example contains the **CREATE** statement for the BOAT_COST UDF that accepts BOAT distinct type, which is defined from a BLOB:

```
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 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.

LOB values as UDF parameters

The user-defined functions (UDFs) can be defined with LOB type parameters and return LOB results. The LOB types can be BLOB, CLOB, or DBCLOB.

About this task

When you pass a LOB value as an argument to a function, the entire LOB value is stored by the database manager before the function is invoked, even if the source of the value is a *LOB locator* host variable. In the following C language application example, either host variable `:clob150K` or `:clob_locator1` is valid as an argument for a function whose corresponding parameter is defined as `CLOB(500K)`:

```
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;
```

The following `CREATE FUNCTION` statement example takes a `CLOB(500K)` parameter:

```
CREATE FUNCTION FINDSTRING (CLOB(500K, VARCHAR(200)))
  ...
```

Both of the following invocations of the FINDSTRING function 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 distinct types that are 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 that is 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.

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.

Before you begin

- 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.

Example

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
```

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.

Before you begin

- 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.

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 preceded 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
```

Index

Special Characters

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X