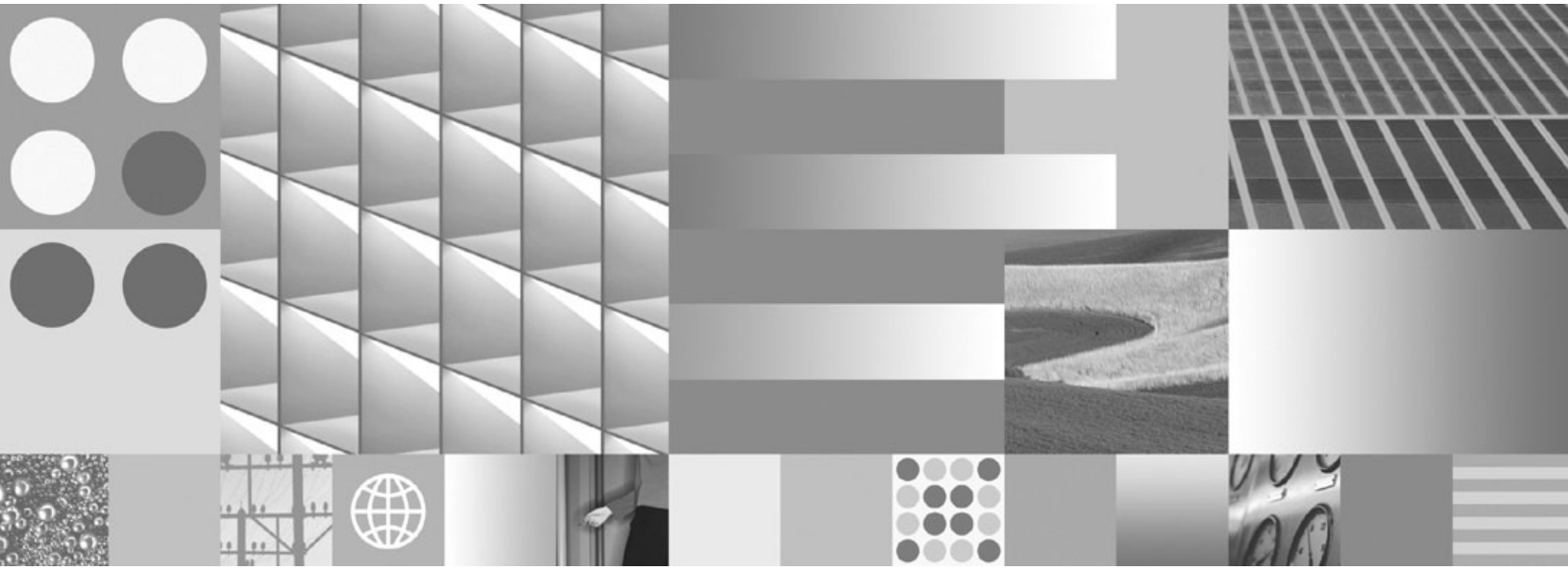
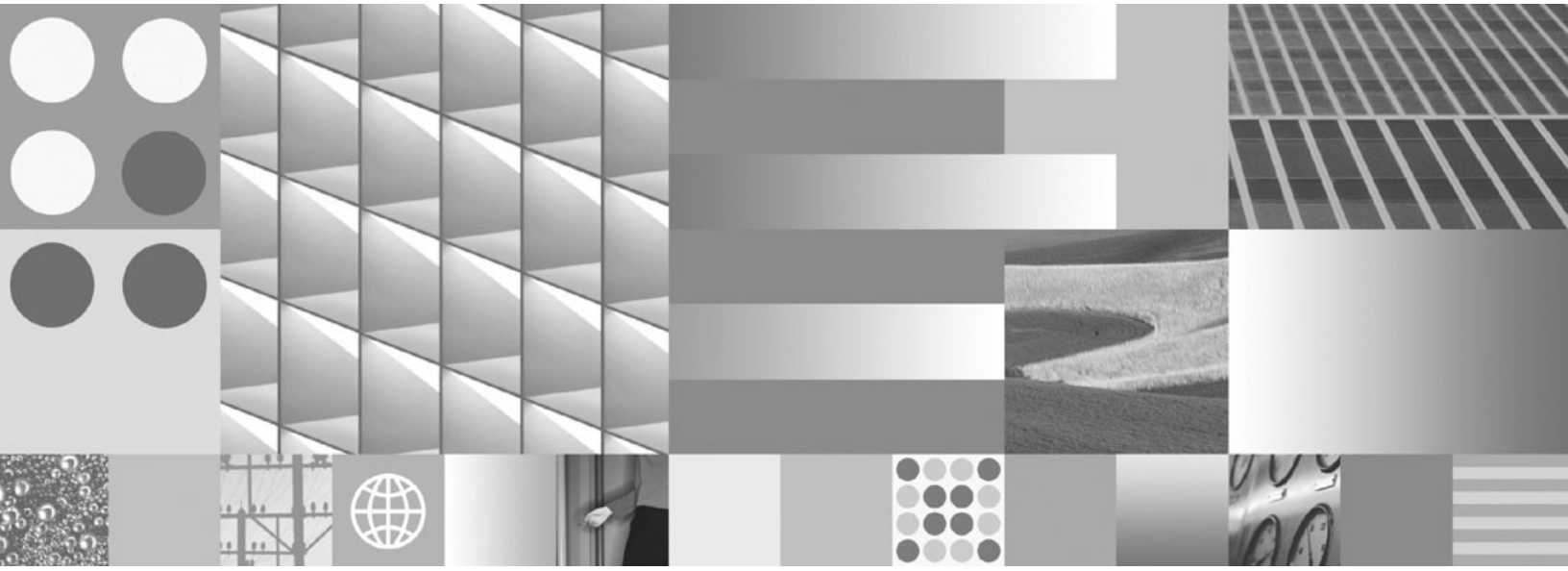
DB2 Version 9.5
for Linux, UNIX, and Windows



Developing User-defined Routines (SQL and External)
Updated March, 2008

SC23-5855-01

DB2 Version 9.5
for Linux, UNIX, and Windows



Developing User-defined Routines (SQL and External)
Updated March, 2008

SC23-5855-01

Note

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 365.

Edition Notice

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

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

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

© Copyright International Business Machines Corporation 1993, 2008. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book vii
Who should use this book vii
How this book is structured vii

Chapter 1. Overview of routines 1
Routines 1
Benefits of using routines 1
Types of routines 3
System-defined and user-defined routines 5
System-defined routines 5
User-defined routines 6
Comparison of system-defined and user-defined routines 7
Determining when to use system-defined or user-defined routines 8
Functional types of routines 8
Routines: Procedures 9
Routines: Functions 10
Routines: Methods 14
Comparison of functional types of routines . . . 15
Determining what functional type of routine to use 18
Implementations of routines 19
Built-in routine implementation 20
Sourced routine implementation 20
SQL routine implementation 20
External routine implementation 21
Supported APIs and programming languages for external routine development 21
Comparison of supported APIs and programming languages for external routine development 22
Comparison of routine implementations 27
Determining what routine implementation to use 29
Usage of routines 30
Administration of databases with system-defined routines 30
Extension of SQL function support with user-defined functions 31
Auditing using SQL table functions 32
Tools for developing routines 34
IBM Data Studio routine development support 35
SQL statements that can be executed in routines . . 35
SQL access levels in routines 40
Determining what SQL statements can be executed in routines 40
Portability of routines 42
Interoperability of routines 42
Performance of routines 43
Security of routines 50
Securing routines 51
Authorizations and binding of routines that contain SQL 52
Data conflicts when procedures read from or write to tables 55

Chapter 2. Developing routines 59

Chapter 3. Overview of SQL routines 61
SQL routines 61
Tools for developing SQL routines 62
SQL routine development in the IBM Data Studio 62
SQL Procedural Language (SQL PL) 62
Inline SQL PL 63
SQL PL in SQL procedures 64
Inline SQL PL and SQL functions, triggers, and compound SQL (dynamic) statements 65
CREATE statements for SQL routines 66
Determining when to use SQL routines or external routines 67
Determining when to use SQL procedures or SQL functions 68
Determining when to use dynamic compound SQL statements or SQL procedures 69
Rewriting SQL procedures as SQL user-defined functions 70

Chapter 4. Developing SQL procedures 73
SQL procedures 73
Features of SQL procedures 73
Designing SQL procedures 74
Parts of SQL procedures 74
Structure of SQL procedures 75
Array support in SQL procedures 77
Parameters in SQL procedures 79
Parameter markers 79
Variables in SQL procedures (DECLARE, SET statements) 85
XML and XQuery support in SQL procedures . . 86
SQLCODE and SQLSTATE variables in SQL procedures 86
Compound statements and scope of variables in SQL procedures 87
Effect of commits and rollbacks on XML parameter and variable values in SQL procedures 88
Cursors in SQL procedures 88
Cursors for XQuery expressions in SQL procedures 89
SQL PL logic elements in the SQL-procedure body 91
Variable related statements in SQL procedures . . 91
Conditional statements in SQL procedures . . . 91
Looping statements in SQL procedures 93
Transfer of control statements in SQL procedures 96
Condition handlers in SQL procedures 99
Returning result sets from SQL procedures . . 100
Receiving procedure result sets in SQL routines 101
Creating SQL procedures 101
Creating SQL procedures from the command line 101

© Copyright IBM Corp. 1993, 2008

Customizing precompile and bind options for SQL procedures 102
Improving the performance of SQL procedures . . 104

Chapter 5. Overview of external routines 111

External routines 111
External routine features 111
External routine creation 112
External routine library and class management . . 113
Deployment of external routine libraries and classes 114
Security of external routine library or class files 114
Resolution of external routine libraries and classes 115
Modifications to external routine library and class files 115
Backup and restore of external routine library and class files 116
External routine library management and performance 116
Supported APIs and programming languages for external routine development 117
Comparison of supported APIs and programming languages for external routine development 117
32-bit and 64-bit support for external routines . . 123
Performance of routines with 32-bit libraries on 64-bit database servers 124
External routine parameter styles 124
Restrictions on external routines 127

Chapter 6. Developing external routines 131

External function and method features 131
External scalar functions 131
External scalar function and method processing model 133
External table functions 133
External table function processing model . . . 134
Table function execution model for Java . . . 135
Scratchpads for external functions and methods 136
Scratchpads on 32-bit and 64-bit operating systems 140
XML data type support in external routines . . . 140
Writing routines 141
Creating external routines 143
.NET common language runtime (CLR) routines 144
Support for external routine development in .NET CLR languages 145
Tools for developing .NET CLR routines . . . 146
Designing .NET CLR routines 146
Creating .NET CLR routines 154
Building .NET CLR routine code 157
Debugging .NET CLR routines 162
Examples of .NET CLR routines 165
C and C++ routines 204
Support for external routine development in C 205
Support for external routine development in C++ 205
Tools for developing C and C++ routines . . . 206
Designing C and C++ routines 206
Include file required for C and C++ routine development (sqludf.h) 207
Parameters in C and C++ routines 208
Supported SQL data types in C and C++ routines 220
SQL data type handling in C and C++ routines 223
Passing arguments to C, C++, OLE, or COBOL routines 230
Graphic host variables in C and C++ routines 242
C++ type decoration 242
Returning result sets from C and C++ procedures 244
Creating C and C++ routines 245
Building C and C++ routine code 247
Building C and C++ routine code using sample bldrtn scripts 247
Building C and C++ routine code from DB2 Command Window 253
Compile and link options for C and C++ routines 254
Building embedded SQL stored procedures in C or C++ with configuration files 264
Building user-defined functions in C or C++ with configuration files (AIX) 265
Rebuilding DB2 routine shared libraries . . . 266
Updating the database manager configuration file 267

Chapter 7. COBOL procedures 269

Support for external procedure development in COBOL 271
Supported SQL data types in COBOL embedded SQL applications 271
Building COBOL routines 274
Compile and link options for COBOL routines 274
Building IBM COBOL routines on AIX 279
Building UNIX Micro Focus COBOL routines 280
Building IBM COBOL routines on Windows . . 281
Building Micro Focus COBOL routines on Windows 282

Chapter 8. Java routines 285

Supported Java routine development software . . 285
JDBC and SQLJ application programming interface support for Java routines 286
Specification of an SDK for Java routine development (UNIX) 286
Specification of a driver for Java routines 287
Tools for developing Java (JDBC and SQLJ) routines 288
Designing Java routines 288
Supported SQL data types in Java routines . . 289
Connection contexts in SQLJ routines 291
Parameters in Java routines 291
Parameter style JAVA procedures 292
Parameter style JAVA Java functions and methods 293
DB2GENERAL routines 294

DB2GENERAL UDFs 294
Supported SQL data types in DB2GENERAL routines 296
Java classes for DB2GENERAL routines . . . 297
DB2GENERAL Java class: COM.IBM.db2.app.StoredProc 298
DB2GENERAL Java class: COM.IBM.db2.app.UDF 299
DB2GENERAL Java class: COM.IBM.db2.app.Lob 301
DB2GENERAL Java class: COM.IBM.db2.app.Blob 301
DB2GENERAL Java class: COM.IBM.db2.app.Clob 301
Passing parameters of data type ARRAY to Java routines 302
Returning result sets from JDBC procedures . . 303
Returning result sets from SQLJ procedures . . 303
Receiving procedure result sets in JDBC applications and routines 304
Receiving procedure result sets in SQLJ applications and routines 305
Restrictions on Java routines 306
Table function execution model for Java . . . 307
Creating Java routines 308
Creating Java routines from the command line 309
Building Java routine code 311
Building JDBC routines 311
Building SQL routines 312
Compile and link options for Java (SQLJ) routines 313
Deploying Java routine class files to DB2 database servers 314
JAR file administration on the database server 315
Updating Java routine classes 316
Examples of Java (JDBC) routines 316
Example: Array data type in Java (JDBC) procedure 316
Example: XML and XQuery support in Java (JDBC) procedure 317

Chapter 9. OLE automation routine design 321
Creating OLE automation routines 321
OLE routine object instances and scratchpad considerations 322
Supported SQL data types in OLE automation . . 322
OLE automation routines in BASIC and C++ . . . 323

Chapter 10. OLE DB user-defined table functions 327
Creating an OLE DB table UDF 327
Fully qualified rowset names 329
Supported SQL data types in OLE DB 330

Chapter 11. Routine invocation 333
Authorizations and binding of routines that contain SQL 334
Routine names and paths 337
Nested routine invocations 338
Invoking 32-bit routines on a 64-bit database server 339
Routine code page considerations 339
References to procedures 340
Procedure selection 340
Calling procedures 341
References to functions 349
Function selection 350
Distinct types as UDF or method parameters 351
LOB values as UDF parameters 352
Invoking scalar functions or methods 352
Invoking user-defined table functions 353

Appendix A. Overview of the DB2 technical information 355
DB2 technical library in hardcopy or PDF format 355
Ordering printed DB2 books 358
Displaying SQL state help from the command line processor 358
Accessing different versions of the DB2 Information Center 359
Displaying topics in your preferred language in the DB2 Information Center 359
Updating the DB2 Information Center installed on your computer or intranet server 360
DB2 tutorials 361
DB2 troubleshooting information 362
Terms and Conditions 362

Appendix B. Notices 365

Index 369

About this book

This book provides information concerning the development of user-defined routines when a system-defined routine that provides the required functionality is not available.

Who should use this book

This book is intended for database architects, database administrators, and application developers of all levels.

- Database architects interested in learning about how routine objects can be created and used to modularize SQL and related logic that can be reused in multiple contexts within the database architecture.
- Database administrators interested in learning how to create, manage, deploy, secure, troubleshoot, and improve system performance by using user-defined routines within a database management system.
- Application developers interested in learning how and when to encapsulate SQL statements and application logic into routines so as to improve application modularity and performance as well as the step by step method of designing, creating, and building user-defined routines. Application developers should have experience writing SQL statements and programming experience using one of the supported programming languages for routine development (C, C++, Java™, COBOL, C#, Visual Basic, or another .NET CLR supported programming language).

How this book is structured

The major subject areas discussed in the chapters of this book are as follows:

Overviews

- Chapter 1, "Overview of routines," presents an overview of system-defined and user-defined routines, the functional types, implementations, usage, and tools to develop routines.
- Chapter 3, "Overview of SQL routines," provides information about SQL routines and how to determine when to use SQL routines, procedures, or functions.
- Chapter 5, "Overview of external routines," presents general information about external routines.

Routine, procedure and function development

- Chapter 2, "Developing routines," generally describes the routine development procedure.
- Chapter 4, "Developing SQL procedures," describes the features, design, and creation of SQL procedures.
- Chapter 6, "Developing external routines," includes descriptions outlining the development of C, C++, and .NET CLR routines.
- Chapter 7, "COBOL procedures," presents information on how to develop COBOL routines.
- Chapter 8, "Java routines," describes the development of Java routines.

© Copyright IBM Corp. 1993, 2008

- Chapter 9, “OLE automation routine design,” provides information for the development of Object Linking and Embedding (OLE) automation routines.
- Chapter 10, “OLE DB user-defined table functions,” describes how to create Object Linking and Embedding (OLE) DB user-defined table functions.

Routine invocation

- Chapter 11, “Routine invocation,” discusses how to invoke the routine after it has been developed.

Chapter 1. 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.

Before deciding to implement a routine, it is important that you understand what routines are, how they are implemented, and how they can be used. The following concept topics will help you gain an understanding of routines so that you can make informed decisions about when and how to use them in your database environment:

- "Benefits of using routines"
- "Types of routines" on page 3
- "Tools for developing routines" on page 34
- "Usage of routines" on page 30
- "SQL access levels in routines" on page 40
- "External routine creation" on page 112
- "Performance of routines" on page 43
- "Security of routines" on page 50

After learning about routines, you might want to perform one of the following tasks:

- "Determining when to use system-defined or user-defined routines" on page 8
- "Determining what functional type of routine to use" on page 18
- "Determining what routine implementation to use" on page 29
- "Tools for developing routines" on page 34

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 system-defined, 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 DB2®, refer to the topic: "Types of routines" on page 3.

The many features of routines are part of why there are so many useful applications of routines.

Benefits of using routines

The following benefits can be gained by using routines:

© IBM Corporation 1993, 2008

Encapsulate application logic that can be invoked from an SQL interface
In an environment containing many different client applications that have common requirements, the effective use of routines can simplify code reuse, code standardization, and code maintenance. If a particular aspect of common application behavior needs to be changed in an environment where routines are used, only the affected routine that encapsulates the behavior requires modification. Without routines, application logic changes are required in each application.

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 CREATE TABLE; however the user can be given permission to invoke routines that contain one or more specific implementations of the statement, thus simplifying privilege management through encapsulation of privileges.

Improve application performance by reducing network traffic
When applications run on a client computer, each SQL statement is sent separately from the client computer to the database server computer to be executed and each result set is returned separately. This can result in high levels of network traffic. If a piece of work can be identified that requires extensive database interaction and little user interaction, it makes sense to install this piece of work on the server to minimize the quantity of network traffic and to allow the work to be done on the more powerful database servers.

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 executed within routines can perform better than if executed in client applications. Routines that are created with the NOT FENCED clause run in the same process as the database manager, and can therefore use shared memory for communication, which can result in improved application performance.

Allow the interoperability of logic implemented in different programming languages
Because code modules might be implemented by different programmers in different programming languages, and because it is generally desirable to reuse code when possible, DB2 routines support a high degree of interoperability.

- Client applications in one programming language can invoke routines that are implemented in a different programming language. For example C client applications can invoke .NET common language runtime routines.
- Routines can invoke other routines regardless of the routine type or routine implementation. For example a Java procedure can invoke an embedded SQL scalar function.
- Routines created in a database server on one operating system can be invoked from a DB2 client running on a different operating system.

The benefits described above are just some of the many benefits of using routines. Using routines can be beneficial to a variety of users including database administrators, database architects, and database application developers. For this reason there are many useful applications of routines that you might want to explore.

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

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:

- "System-defined routines" on page 5
- "User-defined routines" on page 6

The supported functional types of routines are:

- "Routines: Procedures" on page 9 (also called stored procedures)
- "Routines: Functions" on page 10
- "Routines: Methods" on page 14

The supported routine implementations are:

- "Built-in routine implementation" on page 20
- "Sourced routine implementation" on page 20
- "SQL routine implementation" on page 20
- "External routine implementation" on page 21

The following diagram illustrates the classification hierarchy of routines. All routines can be either system-defined 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.

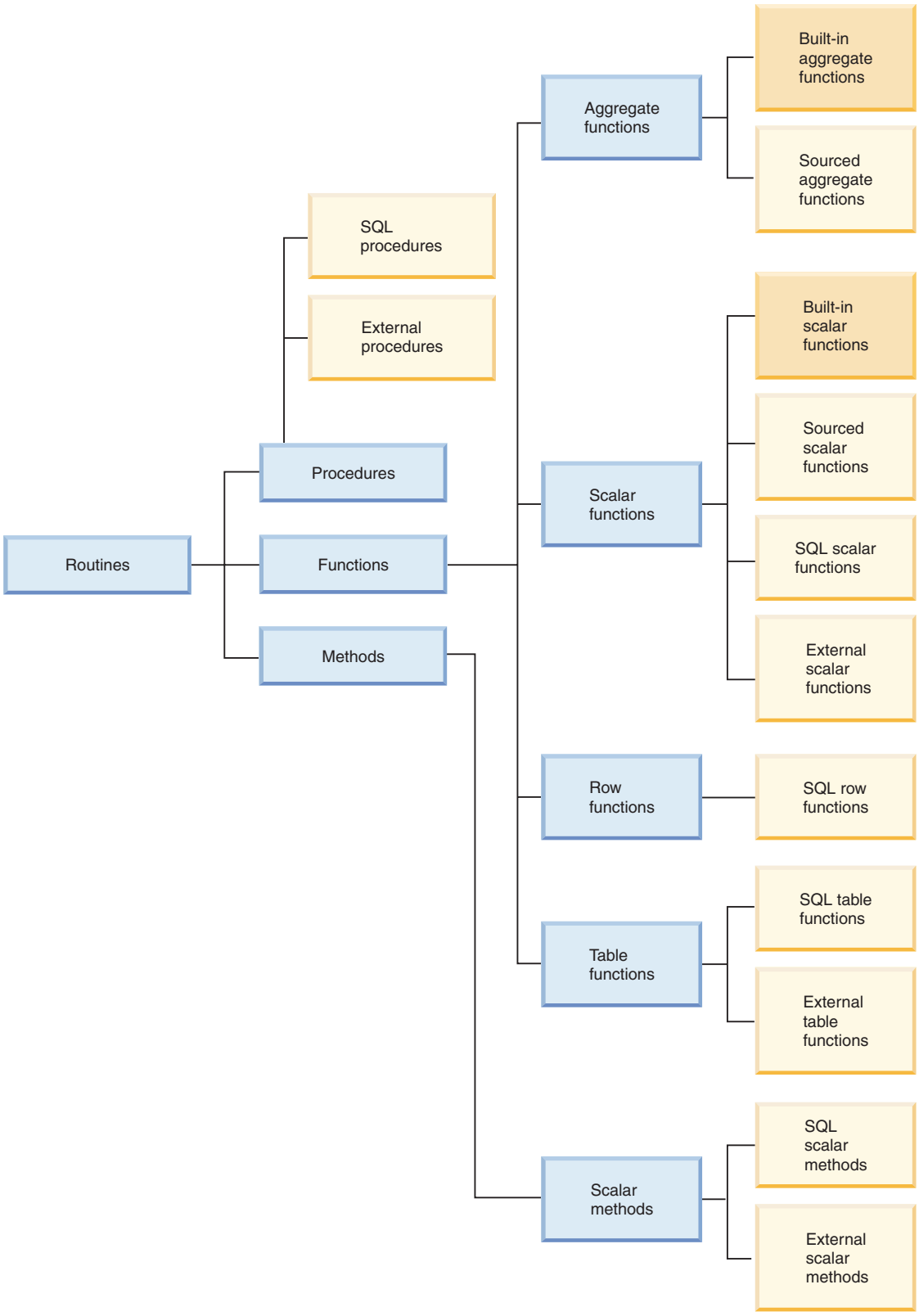


Figure 1. Classifications of routines

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

System-defined and user-defined routines

One of the most straight-forward ways of categorizing routines is to divide them into “System-defined routines” and “User-defined routines” on page 6.

System-defined 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.

System-defined routines

System-defined 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 and system-defined routines, previously called SQL Administrative Routines.

Built-in system-defined 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 or be as secure as built-in routines.

Other system-defined routines that you can use are provided with DB2 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 DB2, they are not built-in routines. Instead they are implemented as pre-installed user-defined routines. These routines typically encapsulate a utility function. Some examples of these include: *SNAP_GET_TAB_V91*, *HEALTH_DB_HI*, *SNAP_WRITE_FILE*, and *REBIND_ROUTINE_PACKAGE*. You can immediately use these functions and procedures, provided that you have the SYSPROC schema and SYSFUN schema in your CURRENT PATH special register. It is a good idea to peruse the set of DB2 provided system-defined routines if you are considering implementing an external routine that performs administrative functionality.

Of particular interest, you might find the *ADMIN_CMD* procedure useful as it provides a standard interface for executing many popular DB2 commands through an SQL interface.

System-defined routines make it faster and easier for you to implement complex SQL queries and powerful database applications since they are ready-to-use.

User-defined routines

DB2 provides routines that capture the functionality of most commonly used arithmetic, string, and casting functions. However, DB2 also allows you to create routines to encapsulate logic of your own. These routines are called user-defined routines. You can create your own "Routines: Procedures" on page 9, "Routines: Functions" on page 10 and "Routines: Methods" on page 14 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 statement" in the *SQL Reference*
- "CREATE FUNCTION statement" in the *SQL Reference*
- "CREATE METHOD statement" in the *SQL Reference*

The clauses specific to each of the CREATE statements define characteristics of the routine, such as the routine name, the number and type of routine arguments, and details about the routine logic. DB2 uses the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in the DB2 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 routine implementation" on page 20: user-defined routines can be *sourced* from the logic of existing built-in routines.
- "SQL routine implementation" on page 20: user-defined routines can be implemented using only SQL statements.
- "External routine implementation" on page 21: 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.

Upon database migration it might be necessary to verify or update routine implementations.

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

User-defined routines can invoke system-defined 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 system-defined routines do not exist.

Comparison of system-defined and user-defined routines

Understanding the differences between system-defined 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.

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

Table 1. Comparison of system-defined and user-defined routines

Characteristic	System-defined 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 system-defined routines within user-defined routines if you want to extend the functionality of the system-defined 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.
Migration	No or little migration impact.	Release to release migrations might require you to verify your routines.
Performance	Perform better than equivalent user-defined routines.	Generally do not perform as well as equivalent system-defined routines.
Stability	Strong type support and error handling.	Type support and error handling must be programmed by the routine developer.

Whenever it is possible to do so, you should choose to use the system-defined 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 system-defined routine performs the specific business logic that you want to implement.

Determining when to use system-defined or user-defined routines

System-defined 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 system-defined routine adequately contains the functionality that you require.

To determine whether to use a system-defined or user-defined routine, do the following:

1. Determine what functionality you want the routine to encapsulate.
2. Check the list of available system-defined routines to see if there are any that meet some or all of your requirements.
   - If there is a system-defined 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 system-defined 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 system-defined 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 system-defined 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 system-defined routine.
   - If no system-defined routine adequately captures the functionality that you want to encapsulate, create a user-defined routine.

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

- "Routines: Procedures" on page 9, 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.
-

"Routines: Functions" on page 10 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.

- "Routines: Methods" on page 14 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.
  - Refer to the topic ″Restrictions on procedures″

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 system-defined 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
    - Refer to the topic, "Supported external routine implementation programming languages".

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" in *SQL Reference, Volume 1*
- 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 Database for System 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 system-defined 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 are easy to implement. External methods provide support for flexible logic implementation, allow a user to develop method logic in their preferred programming language, and provide the ability to store intermediate values.

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.
- Refer to: Restrictions on scalar functions.

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

– Refer to the topic, “Comparison of supported APIs and programming languages for external routine development” on page 22.

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.
- Refer to the topic: ″Restrictions on row functions″

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 qould 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.
- Refer to the topic, ″Restrictions on table functions″.

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 system-defined 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 20
- “External routine implementation” on page 21: 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.

"Routines: Procedures" on page 9, "Routines: Functions" on page 10, and "Routines: Methods" on page 14 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 system defined routines of this type?	• Yes, many. • See the SQL reference for a list of system-defined procedures.	• Yes, many. • See the SQL reference for a list of system-defined 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. 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.

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 15

     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" on page 35
   - 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.

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.

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 routines
- Sourced routines
- SQL routines
- External routines

System-defined routines 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 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 27

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 DB2 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 DB2.

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, or where restricted a subset of SQL PL, can be used to create SQL procedures, functions, and methods. SQL routine implementations can contain traditional SQL statements as well as SQL PL statements, although depending on the functional type of the routine, the execution of some SQL statements might be restricted.

In general SQL procedures support more features than SQL functions including a powerful and easy to implement error-handling mechanism. SQL routines cannot directly make system calls and cannot directly perform operations on entities that reside outside of the database.

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.

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 DB2 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 email.

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 DB2 external routines (procedures and functions) using the following APIs and associated programming languages:

- 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 implementation go quickly. • SQL Procedural Language (SQL PL) elements allow for control-flow logic around SQL operations and queries. • Strong data type support.	• 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 DB2 instances, however these routines will not perform as well as 64-bit COBOL procedures within a 64-bit DB2 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	• 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 DB2 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	• Strong data type support. • 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.	• Strong data type support. • 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.	• Strong data type support. • 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 DB2 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.

Whenever possible, if there is an existing system-defined 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, follow the procedure below. It is important to determine what routine implementation to use before proceeding too far with routine design.

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

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 15
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 40
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. Some examples of scenarios in which you might use routines are listed below:

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

Administration of databases with system-defined routines

Administering databases through applications is possible and has become easier with the introduction of system-defined routines for the explicit purpose of doing administrative functions. As of Version 8.1, DB2 provides a set of system-defined procedures and functions in the SYSPROC, SYSFUN, and SYSTOOLS schemas that are ready-to-use for doing administrative tasks, including the execution of DB2 commands through an SQL interface, modification of configuration parameters, package management, snapshot related tasks, and more. You might choose to use system-defined administrative routines if you require an application to perform administrative tasks or if you want to access the results of administrative tasks

through an SQL interface so that you filter, sort, modify, or reuse the results in another query and if you don't want to create your own routines to do this.

As of DB2 Universal Database™ Version 9.1 there is a new system-defined administrative routine, named the ADMIN_CMD. It along with the many other system-defined routines provide comprehensive administration support.

ADMIN_CMD for invoking DB2 commands through a SQL interface

As of Version 9.1, there is a new system-defined administrative routine called the ADMIN_CMD that allows you to execute DB2 commands through an SQL interface. Essentially this routine allows you to pass in as an argument a DB2 command with appropriate flags and values as a string parameter. The routine executes the string containing the DB2 command and returns the results in a tabular or scalar format that can be used as part of a larger query or operation. This functionality makes it easier than ever to write administrative database applications.

System-defined administrative routines

Examples of other system-defined routines include: SNAPSHOT_TABLE, HEALTH_DB_HI, SNAPSHOT_FILEW, REBIND_ROUTINE_PACKAGE. These and many more system-defined routines can be used from the CLP or in database applications wherever invocation of the specified routine is supported.

The ADMIN_CMD routine and the other system-defined routines are available for use provided that you have the SYSPROC schema and SYSFUN schema names included in your CURRENT PATH value, which they are by default.

For examples of how to use the system-defined routines, refer to system-defined routine specific reference documentation.

Extension of SQL function support with user-defined functions

If no system-defined 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. 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 system-defined 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 email whenever a change is made to a particular column in a table. Such a function is not available within the set of system-defined 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.

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 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 DB2 Database for Linux®, UNIX®, and Windows® 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 graphical user-interface (GUI) tool, provided with DB2, is available for developing routines in DB2 database servers:

- IBM® Data Studio

The following command line interface, provided with DB2, is available for developing routines in DB2 database servers:

- DB2 Command Line Processor (DB2 CLP)

Several IBM software products provide graphical tools for developing routines in DB2 database servers including, but not limited to:

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

Several open source software products provide graphical tools for developing routines in DB2 database servers including, but not limited to:

- DB2 Web Tools for the Eclipse Framework

Some DB2 features can be used to add graphical tool support for developing routines from within software provided by other vendors, including:

- IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2

There are no tools available for developing methods.

IBM Data Studio routine development support

The IBM Data Studio provides an easy-to-use development environment for creating, building, debugging, testing, and deploying stored procedures. The 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 DB2 server. Additionally, with the developer tool, you can develop stored procedures on one operating system and build them on other server operating systems.

The developer tool is a graphical application that supports rapid development. Using the developer tool, you can perform the following tasks:

- Create new stored procedures.
- Build stored procedures on local and remote DB2 servers.
- Modify and rebuild existing stored procedures.
- Test and debug the execution of installed stored procedures.

The Data Studio is installed from a DVD that is included with DB2 Database for Linux, UNIX, and Windows. After you install Data Studio, you can launch it from the following locations:

- On Windows: Click **Start** > **All Programs** > **IBM Software Development Platform** > **IBM Data Studio** > **IBM Data Studio**
- On Linux: Click **Programming** > **IBM Data Studio** or **Development** > **IBM Data Studio**, depending on your Linux version.

You can also launch Data Studio from the Control Center for DB2 for OS/390®. You can start Data Studio as a separate process from the Control Center Tools menu, toolbar, or Stored Procedures folder. In addition, from the Data Studio Project window, you can export one or more selected SQL stored procedures built to a DB2 for OS/390 server to a specified file capable of running within the command line processor (CLP).

The Data Studio allows you to manage your work in projects. Each Data Studio project saves your connections to specific databases, such as DB2 for OS/390 servers. In addition, you can create filters to display subsets of the stored procedures on each database. When opening a new or existing Data Studio project, you can filter stored procedures so that you view them based on their name, schema, language, or collection ID (for OS/390 only).

SQL statements that can be executed in routines

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.

The following table lists all supported SQL statements, including SQL PL control-statements, and identifies if you can execute each SQL statement within the various types of routines. For each SQL statement listed in the first column, in each of the subsequent columns identifying the supported routine types and implementations, an X indicates if the statement is executable within the routine. A separate column identifies the minimum required SQL access level that must be specified for the statement execution to be allowed. Unless otherwise noted in a footnote, all of the SQL statements can be executed both statically and dynamically.

Table 5. SQL statements that can be executed in routines

SQL statement	Executable in SQL procedures	Executable in SQL functions	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				
BEGIN DECLARE SECTION			X	X	NO SQL(1)
CALL	X	X	X	X	READS SQL DATA
CASE	X				CONTAINS SQL
CLOSE	X		X	X	READS SQL DATA
COMMENT ON	X		X	X	MODIFIES SQL DATA
COMMIT	X(4)		X(4)		MODIFIES SQL DATA
Compound SQL	X	X	X	X	CONTAINS SQL
CONNECT(2)					

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

SQL statement	Executable in SQL procedures	Executable in SQL functions	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
CREATE {ALIAS, BUFFERPOOL, DATABASE PARTITION GROUP, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING, INDEX, INDEX EXTENSION, METHOD, NICKNAME, PROCEDURE, SCHEMA, SEQUENCE, SERVER, TABLE, TABLESPACE, TRANSFORM, TRIGGER, TYPE, TYPE MAPPING, USER MAPPING, VIEW, WRAPPER }	X (6)		X		MODIFIES SQL DATA
DECLARE CURSOR	X		X		NO SQL(1)
DECLARE GLOBAL TEMPORARY TABLE	X		X	X	MODIFIES SQL DATA
DELETE	X	X	X	X	MODIFIES SQL DATA
DESCRIBE(7)			X	X	READS SQL DATA
DISCONNECT(2)					
DROP	X(6)		X	X	MODIFIES SQL DATA
END DECLARE SECTION			X	X	NO SQL(1)
EXECUTE	X		X	X	CONTAINS SQL(3)
EXECUTE IMMEDIATE	X		x	X	CONTAINS SQL(3)
EXPLAIN	X		X	X	MODIFIES SQL DATA
FETCH	X		X	X	READS SQL DATA

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

SQL statement	Executable in SQL procedures	Executable in SQL functions	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
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	X	CONTAINS SQL
GET DIAGNOSTICS	X	X			READS SQL DATA
GOTO	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				CONTAINS SQL
MERGE	X	X	X	X	MODIFIES SQL DATA
OPEN	X		X	X	READS SQL DATA(5)
PREPARE	X		X	X	CONTAINS SQL
REFRESH TABLE			X	X	MODIFIES SQL DATA
RELEASE					
RELEASE SAVEPOINT	X		X	X	MODIFIES SQL DATA
RENAME TABLE			X	X	MODIFIES SQL DATA
RENAME TABLESPACE			X	X	MODIFIES SQL DATA
REPEAT	X				CONTAINS SQL
RESIGNAL	X				MODIFIES SQL DATA
RETURN	X				CONTAINS SQL
REVOKE			X	X	MODIFIES SQL DATA
ROLLBACK(4)	X		X		

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

SQL statement	Executable in SQL procedures	Executable in SQL functions	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
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	X	READS SQL DATA(5)
SET CONNECTION(2)					
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
UPDATE	X	X	X		MODIFIES SQL DATA
VALUES INTO	X		X	X	READS SQL DATA
WHENEVER	X		X		NO SQL(1)
WHILE	X	X			

Note:

1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. Connection management statements are not allowed in any routine execution context.
3. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is 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.
4. 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, COMMIT or ROLLBACK of a unit of work is not allowed.)
5. 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.
6. SQL procedures can only issue CREATE and DROP statements for indexes, tables, and views.

7. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command.

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 below:

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	MODIFIES 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.

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 restrictions below.
- 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.

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.

To determine what SQL statements can be invoked in a particular routine, do the following:

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.

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 DB2 routines are highly portable between operating systems and even between the various DB2 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 DB2 Database for Linux, UNIX, and Windows 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, DB2 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. DB2 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 .NET common language runtime routines.
- A routine can invoke another routine regardless of the routine type or the implementation language of the routine.
  - For example a Java procedure (one type of routine) can invoke an SQL scalar function (another type of routine with a different implementation language).
- A routine created in a database server on one operating system can be invoked from a DB2 client running on a different operating system.

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

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: system-defined or user-defined	• For equivalent logic, built-in routines perform the best, followed by system-defined 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 DB2 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 since 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, DB2 attempts to choose a threaded Java fenced mode process with a Java heap that is large enough to run the routine. Failure to isolate large heap consumers in their own process can result in out-of-Java-heap errors in 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. – To perform their best C and C++ routines should be compiled in 32-bit format if they will be deployed to a 32-bit DB2 instance and in 64-bit format if they will be deployed to a 64-bit DB2 instance. • 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 DB2 from padding parameters with spaces before passing the parameter and decreases the amount of time required to transmit the parameter across a network. For example, if your client application passes the string "A SHORT STRING" to a routine that expects a CHAR(200) parameter, DB2 has to pad the parameter with 186 spaces, null-terminate the string, then send the entire 200 character string and null-terminator across the network to the routine. In comparison, passing the same string, *"A SHORT STRING"*, to a routine that expects a VARCHAR(200) parameter results in DB2 simply passing the 14 character string and a null terminator across the network.
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.

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

Performance consideration	Performance recommendation
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 DB2 database manager, is preferable over creating it with the FENCED clause, which makes it run in a special DB2 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, DB2 requires that a user have a special privilege, CREATE_NOT_FENCED_ROUTINE in order to create NOT FENCED routines. • If an abnormal termination occurs while you are running a NOT FENCED routine, the database manager will attempt an appropriate recovery if the routine is registered as NO SQL. However, for routines not defined as NO SQL, the database manager will fail. • NOT FENCED routines must be precompiled with the WCHARTYPE NOCONVERT option if the routine uses GRAPHIC or DBCLOB data.

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

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.

Prerequisites

- 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 DBADM or SYSADM authorities.

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

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

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

Package Owner

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

Routine Definer

The ID that issues the CREATE statement to register a routine. The routine definer is generally a DBA, but is also often the routine package owner.

When a routine is invoked, at package load time, the authorization to run the routine is checked against the definer's authorization to execute the package or packages associated with the routine (not against the authorization of the routine invoker). For a routine to be successfully invoked, the routine definer must have one of:

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

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

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

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

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

Routine Invoker

The ID that invokes the routine. To determine which users will be invokers of a routine, it is necessary to consider how a routine can be invoked. Routines can be invoked from a command window or from within an embedded SQL application. In the case of methods and UDFs the routine reference will be embedded in another SQL statement. A procedure is invoked by using the CALL statement. For dynamic SQL in an application, the invoker is the runtime authorization ID of the immediately higher-level routine or application containing the routine invocation (however, this ID can also depend on the DYNAMICRULES option with which the higher-level routine or application was bound). For static SQL, the invoker is the value of the OWNER precompile/BIND option of the package that contains the reference to the routine. To successfully invoke the routine, these users will require EXECUTE privilege on the routine. This privilege can be granted by any user with EXECUTE WITH GRANT OPTION privilege on the routine (this includes the routine definer unless the privilege has been explicitly revoked), SYSADM or DBADM authority by explicitly issuing a GRANT statement.

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

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:

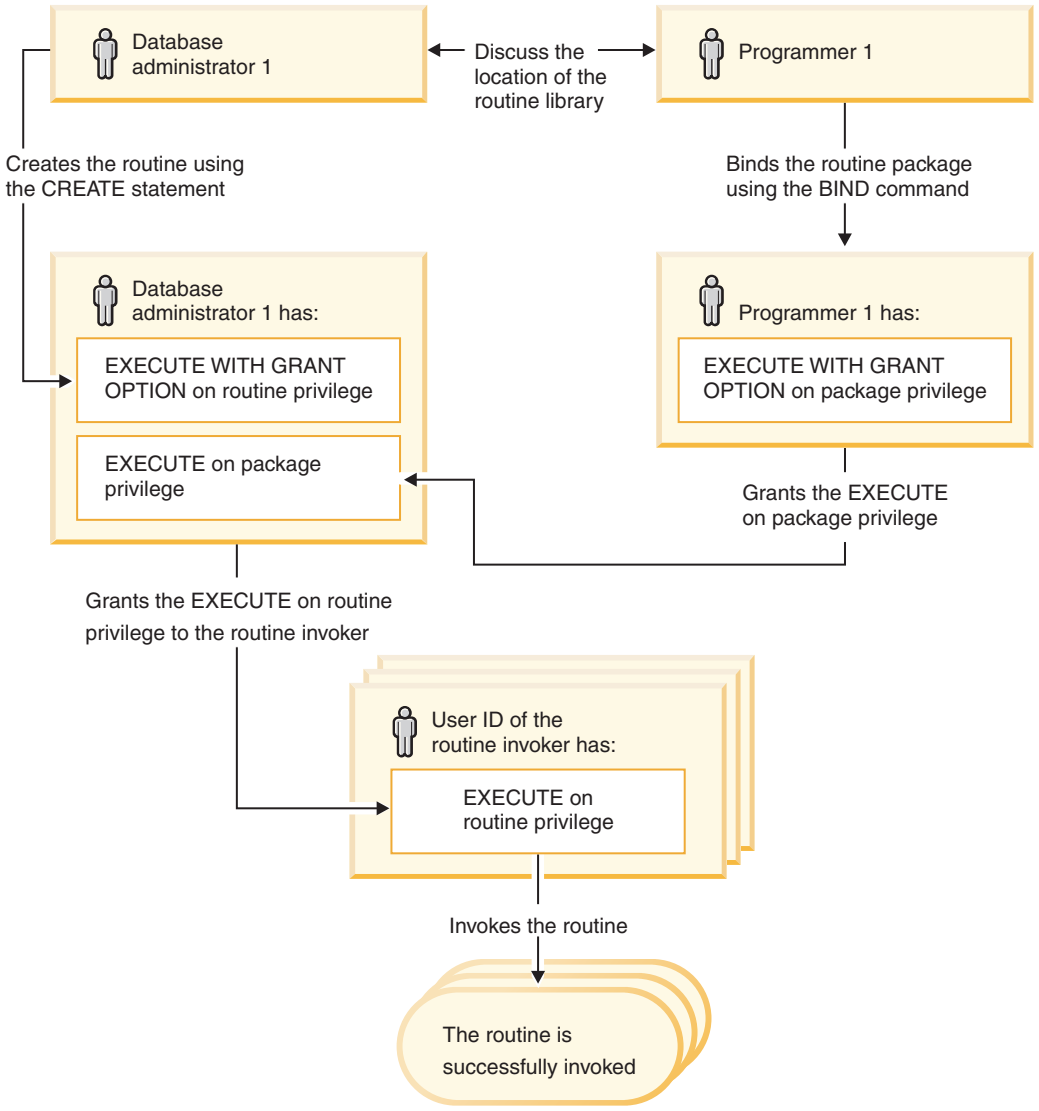


Figure 2. Managing the EXECUTE privilege on routines

1. Definer performs the appropriate CREATE statement to register the routine. This registers the routine in DB2 with its intended level of SQL access,

establishes the routine signature, and also points to the routine executable. The definer, if not also the package owner, needs to communicate with the package owners and authors of the routine programs to be clear on where the routine libraries reside so that this can be correctly specified in the EXTERNAL clause of the CREATE statement. By virtue of a successful CREATE statement, the definer has EXECUTE WITH GRANT privilege on the routine, however the definer does not yet have EXECUTE privilege on the packages of the routine.

2. Definer must grant EXECUTE privilege on the routine to any users who are to be permitted use of the routine. (If the package for this routine will recursively call this routine, then this step must be done before the next step.)
3. Package owners precompile and bind the routine program, or have it done on their behalf. Upon a successful precompile and bind, the package owner is implicitly granted EXECUTE WITH GRANT OPTION privilege on the respective package. This step follows step one in this list only to cover the possibility of SQL recursion in the routine. If such recursion does not exist in any particular case, the precompile/bind could precede the issuing of the CREATE statement for the routine.
4. Each package owner must explicitly grant EXECUTE privilege on their respective routine package to the definer of the routine. This step must come at some time after the previous step. If the package owner is also the routine definer, this step can be skipped.
5. Static usage of the routine: the bind owner of the package referencing the routine must have been given EXECUTE privilege on the routine, so the previous step must be completed at this point. When the routine executes, DB2 verifies that the definer has the EXECUTE privilege on any package that is needed, so step 3 must be completed for each such package.
6. Dynamic usage of the routine: the authorization ID as controlled by the DYNAMICRULES option for the invoking application must have EXECUTE privilege on the routine (step 4), and the definer of the routine must have the EXECUTE privilege on the packages (step 3).

Data conflicts when procedures read from or write to tables

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

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

top-level statement
A top-level statement is any SQL statement issued from an application, or from a stored procedure that was invoked as a top-level statement. If a procedure is invoked within a dynamic compound statement or a trigger, the compound statement or the statement that causes the firing of the trigger is the top-level statement. If an SQL function or an SQL method contains a nested CALL statement, the statement invoking the function or the method is the top-level statement.

table access context

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

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

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

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

The following rules are applied:

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

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

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

Suppose an application issues the statement:

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

UDF1 contains the statements:

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

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

The following does not result in a conflict:

Suppose an application issues the statements:

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

UDF2 contains the statements:

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

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

Chapter 2. Developing routines

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

There are different functional types of routines and routine implementations, however the basic steps for developing routines are generally 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.

Depending on what type of routine you choose to develop there are some specific procedures you must follow. This topic will direct you to the appropriate topics for getting started with routine development.

Prerequisites

- Read and understand basic routine concepts:
  - To learn about types of routines, useful applications of routines, tools for developing routines, routine best practices and more, see the topic:
    - Chapter 1, "Overview of routines," on page 1
- Learn about the available routine development tools that make it faster and easier to develop routines:
  - To learn about the available tools for routine development, see the topic:
    - "Tools for developing routines" on page 34

Procedure

1. Determine if an existing system-defined routine already meets your routine needs.
   - If a system-defined routine meets your needs, you might want to refer to Chapter 11, "Routine invocation," on page 333.
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" on page 61.
   - If an external routine is required, refer to the information about "External routines" on page 111.

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 you must execute a routine functional type specific CREATE statement. 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. DB2 uses the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in DB2's system 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.

© IBM Corporation 1993, 2008

Because external routines have their logic implemented in user-created libraries or classes 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 might want to:

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

Chapter 3. Overview of SQL routines

SQL routines are routines that have logic implemented with only SQL statements, including SQL Procedural Language (SQL PL) statements. They are characterized by having their routine-body logic contained within the CREATE statement that is used to create them. You can create SQL procedures, SQL functions, and SQL methods. Although they are all implemented in SQL, each routine functional type has different features.

Before deciding to implement a SQL routine, it is important that you first understand what SQL routines are, how they are implemented, and used by reading an "Overview of routines". With that knowledge you can then learn more about SQL routine from the following concept topics so that you can make informed decisions about when and how to use them in your database environment:

- SQL procedures
- SQL functions
- Tools for developing SQL routines
- SQL Procedural Language (SQL PL)
- Comparison of SQL PL and inline SQL PL
- SQL PL statements and features
- Supported inline SQL PL statements and features
- Determining when to use SQL procedures or SQL functions
- Restrictions on SQL routines

After having learned about SQL routines, you might want to do one of the following tasks:

- Develop SQL procedures
- Develop SQL functions
- Develop SQL methods

SQL routines

SQL routines are routines that have logic implemented with only SQL statements, including SQL Procedural Language (SQL PL) statements. They are characterized by having their routine-body logic contained within the CREATE statement that is used to create them. This is in contrast with external routines that have their routine logic implemented in a library built form programming source code. In general SQL routines can contain and execute fewer SQL statements than external routines; however they can be every bit as powerful and high performing when implemented according to best practices.

You can create SQL procedures, SQL functions, and SQL methods. Although they are all implemented in SQL, each routine functional type has different features.

© Copyright IBM Corp. 1993, 2008

Tools for developing SQL routines

SQL routine development tools make it faster and easier to create SQL procedures. The following GUI and command line tools can be used to create SQL procedures:

- DB2 tools
  - Command Line Processor (CLP)
  - IBM Data Studio
  - IBM Database Development Add-In for Visual Studio .NET 1.2
- IBM Software Group and IBM Information Management Tools
  - IBM Rational Application Developer
  - IBM WebSphere Application Server

SQL routine development in the IBM Data Studio

The IBM Data Studio is a separately installed development environment that provides support for SQL routine development. Within the Data Studio you will find tools and documentation support for performing the following SQL routine development tasks:

- Creating SQL routines
- Creating SQL routines that will work on DB2 for z/OS database servers
- Modifying SQL routines
- Debugging SQL procedures
- Comparing SQL routine definitions within a project or on a database server
- Deploying SQL routines to DB2 database servers

You can also do many more things within the Data Studio that can assist you in developing SQL routines, including: querying, modifying, loading, and extracting table data, working with XML functions, developing Java routines, and more!

SQL Procedural Language (SQL PL)

The SQL Procedural Language (SQL PL) is a set of SQL statements that was introduced in DB2 Universal Database Version 7 to provide the procedural constructs necessary for implementing control flow logic around traditional SQL queries and operations. SQL PL has evolved since then and the current set of SQL PL statements and language features provides support for comprehensive high-level programming in SQL.

SQL PL is a subset of the SQL Persistent Stored Modules (SQL/PSM) language standard. The specification of the current SQL/PSM standard can be found in ANSI/ISO/IEC 9075-4:1999 Information Technology, Database Language SQL, Part 4: Persistent Stored Modules (SQL/PSM).

SQL PL has a simple syntax that includes support for variables, conditional statements, looping statements, transfer of control statements, error management statements, and result set manipulation statements. You can use SQL PL in a variety of contexts including in routines with SQL implementations, and a subset of SQL PL statements can be used in triggers, and dynamic compound SQL statements.

SQL PL consists of the following statements:

- Variable related statements
  - DECLARE <variable> DEFAULT <value>

  - DECLARE <condition>
  - DECLARE <condition handler>
  - DECLARE CURSOR
  - SET (assignment-statement)
- Conditional statements
  - CASE (2 forms)
  - IF
- Looping statements
  - FOR
  - LOOP
  - REPEAT
  - WHILE
- Transfer of control statements
  - CALL
  - GOTO
  - ITERATE
  - LEAVE
  - RETURN
- Error management statements
  - SIGNAL
  - RESIGNAL
- Result set manipulation statements
  - ASSOCIATE LOCATOR(S)
  - ALLOCATE CURSOR

Inline SQL PL

Inline SQL PL is a subset of SQL PL features that that can be used within atomic dynamic compound SQL statements that are used in triggers, SQL functions, SQL methods, and standalone code. Standalone code refers to SQL PL scripting.

Inline SQL PL is described as ″inline″, because the logic is expanded into and executed with the SQL statements that reference them.

The following SQL PL statements are considered to be part of the set of inline SQL PL statements:

- Variable related statements
  - DECLARE <variable>
  - DECLARE <condition>
  - SET statement (assignment statement)
- Conditional statements
  - IF
  - CASE expression
- Looping statements
  - FOR
  - WHILE
- Transfer of control statements
  - GOTO

  - ITERATE
  - LEAVE
  - RETURN
- Error management statements
  - SIGNAL
  - GET DIAGNOSTICS

Other SQL PL statements that are supported in SQL procedures are not supported in dynamic compound statements. Cursors and condition handlers are not supported in inline SQL PL and therefore neither is the RESIGNAL statement.

Because inline SQL PL statements must be executed in dynamic compound statements, there is no support for PREPARE, EXECUTE, or EXECUTE IMMEDIATE statements.

Also, because they must be used in an atomic dynamic compound SQL statement, all or none of the member statements must commit successfully. Therefore the COMMIT and ROLLBACK statements are not supported either.

As for the LOOP and REPEAT statements, the WHILE statement can be used to implement equivalent logic.

Although triggers and SQL functions only support the inline SQL PL statements, it is possible to indirectly make use of the larger set of SQL PL statement, by calling an SQL procedure from these interfaces.

Standalone scripting with inline SQL PL consists of executing a dynamic compound statement within a Command Line Processor (CLP) script or directly from a CLP prompt. Dynamic compound SQL statements are bounded by the keywords BEGIN ATOMIC and END and must end with a non-default terminator character. They can contain SQL PL and other SQL statements.

Because inline SQL PL statements are expanded within the SQL statements that reference them rather than being individually compiled, there are some minor performance considerations that should be considered when you are planning on whether to implement your procedural logic in SQL PL in an SQL procedure or with inline SQL PL in a function, trigger, or dynamic compound statement.

SQL PL in SQL procedures

SQL PL statements are primarily used in SQL procedures. SQL procedures can contain basic SQL statements for querying and modifying data, but they can also include SQL PL statements for implementing control flow logic around the other SQL statements. The complete set of SQL PL statements can be used in SQL procedures.

SQL procedures also support parameters, variables, assignment statements, a powerful condition and error handling mechanism, nested and recursive calls, transaction and savepoint support, and the ability to return multiple result sets to the procedure caller or a client application.

SQL PL, when used within SQL procedures, allows you to effectively program in SQL. The high-level language of SQL PL, strong typing of SQL, and the additional features that SQL procedures provide makes programming with SQL PL fast and easy to do.

As a simple example of SQL PL statements being used in a SQL procedure, consider the following example:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
        INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
   UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
        WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
      SET salary = salary * 1.05, bonus = 500
        WHERE empno = empNum;
  ELSE
    UPDATE employee
      SET salary = salary * 1.03, bonus = 0
        WHERE empno = empNum;
  END IF;
END
```

Inline SQL PL and SQL functions, triggers, and compound SQL (dynamic) statements

Inline SQL PL statements can be executed from compound SQL (dynamic) statements, SQL functions, and triggers.

A compound SQL (dynamic) statement is one that allows you to group multiple SQL statements into an optionally atomic block in which you can declare variables, and condition handling elements. These statements are compiled by DB2 as a single SQL statement and can contain inline SQL PL statements.

The bodies of SQL functions and triggers can contain compound SQL statements and can also include some inline SQL PL statements.

On their own, compound SQL statements are useful for creating short scripts that perform small units of logical work with minimal control flow, but that have significant data flow. Within functions and triggers, they allow for more complex logic to be executed when those objects are used.

As an example of a dynamic compound SQL statement that contains SQL PL, consider the following:

```
BEGIN ATOMIC
  FOR row AS
    SELECT pk, c1, discretize(c1) AS v FROM source
  DO
    IF row.v is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
END
```

The compound statement is bounded by the keywords BEGIN and END. It includes use of both the FOR and IF/ELSE control-statements that are part of SQL PL. The FOR statement is used to iterate through a defined set of rows. For each row a column's value is checked and conditionally, based on the value, a set of values is inserted into another table.

As an example of a trigger that contains SQL PL, consider the following:

```
CREATE TRIGGER validate_sched
NO CASCADE BEFORE INSERT ON c1_sched
FOR EACH ROW
MODE DB2SQL
Vs: BEGIN ATOMIC

  IF (n.ending IS NULL) THEN
    SET n.ending = n.starting + 1 HOUR;
  END IF;

  IF (n.ending > '21:00') THEN
    SIGNAL SQLSTATE '80000'  SET MESSAGE_TEXT =
           'Class ending time is after 9 PM';
  ELSE IF (n.DAY=1 or n.DAY-7) THEN
    SIGNAL SQLSTATE '80001' SET MESSAGE_TEXT =
           'Class cannot be scheduled on a weekend';
  END IF;
END vs;
```

This trigger is activated upon an insert to a table named c1_sched and uses SQL PL to check for and provide a class end time if one has not been provided and to raise an error if the class end time is after 9 pm or if the class is scheduled on a weekend. As an example of a scalar SQL function that contains SQL PL, consider the following:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

This simple function returns a scalar price value, based on the value of an input parameter that identifies a vendor. It also uses the IF statement.

For more complex logic that requires output parameters, the passing of result sets or other more advanced procedural elements SQL procedures might be more appropriate.

CREATE statements for SQL routines

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

SQL procedures, functions, and methods each have a respective CREATE statement. Although the syntax for these statements is different, there are some common elements to them. In each you must specify the routine name, and parameters if there are to be any as well as a return type. You can also specify additional keywords that provide DB2 with information about the logic contained

in the routine. DB2 uses the routine prototype and the additional keywords to identify the routine at invocation time, and to execute the routine with the required feature support and best performance possible.

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

Determining when to use SQL routines or external routines

When implementing routine logic you can choose to implement SQL routines or external routines. There are reasons for choosing each of these two implementations.

To determine when to choose to implement an SQL routine or an external routine, read the following to determine what if any factors might limit your choice.

- Choose to implement SQL routines if:
  - SQL PL and SQL statements provide adequate support to implement the logic that you require.
  - The routine logic consists primarily of SQL statements that query or modify data and performance is a concern. Logic that contains a relatively small amount of control-flow logic relative to the number of SQL statements that query or modify database data will generally perform better with an SQL routine implementation. SQL PL is intended to be used for implementing procedural logic around database operations and not primarily for programming complex logic.
  - The SQL statements that you need to execute can be executed in an external routine implementation.
  - You want to make the modules highly portable between operating system environments and minimize the dependency on programming language code compilers and script interpreters.
  - You want to implement the logic quickly and easily using a high level programming language.
  - You are more comfortable working with SQL than with scripting or programming languages.
  - You want to secure the logic within the database management system.
  - You want to minimize routine maintenance and routine package maintenance upon release migration or operating system upgrades.
  - You want to minimize the amount of code required to implement the logic.
  - You want to make use of strong data typing and minimize the risk of data type conversion errors.
  - You want to maximize the safety of the code that is implemented by minimizing the risk of memory management, pointer manipulation, or other common programming pitfalls.
  - ou want to benefit from special SQL caching support made available when SQL PL is used.
- Choose to implement an external procedure if:
  - If the routine logic is very complex and consists of few SQL statements and routine performance is a concern. Logic such as a complex math algorithm, that involves a large amount of string manipulation, or that does not access the database will generally perform better with an external routine implementation.

- If the SQL statements that you need to execute can be executed in an external routine implementation.
- The routine logic will make operating system calls - this can only be done with external routines.
- The routine logic must read from or write to files - this can only be done with external routines.
- Write to the server file system. Do this only with caution.
- Invoke an application or script that resides on the database server.
- Issue particular SQL statements that are not supported in SQL procedures.
- You are more comfortable programming in a programming language other than SQL PL.

By default if SQL routines can meet your needs, use them. Generally it is a requirement to implement complex logic or to access files or scripts on the database server that motivates the decision to use external routines. Particularly since SQL PL is fast and easy to learn and implement.

Determining when to use SQL procedures or SQL functions

When faced with the choice of implementing logic with SQL PL in an SQL procedure or an SQL function, there are reasons for choosing each of these two implementations.

Read the following to determine when to choose to use an SQL procedure or an SQL function.

Choose to implement an SQL function if:

- Functional requirements can be met by an SQL function and you don't anticipate later requiring the features provided by an SQL procedure.
- Performance is a priority and the logic to be contained in the routine consists only of queries or returns only a single result set.

  When they only contain queries or the return of a single result set an SQL function performs better than a logically equivalent SQL procedure, because of how SQL functions are compiled.

  In SQL procedures, static queries in the form of SELECT statements and full-select statements are compiled individually, such that each query becomes a section of a query access plan in a package when the SQL procedure is created. There is no recompilation of this package until the SQL procedure is recreated or the package is rebound to the database. This means that the performance of the queries is determined based on information available to the database manager at a time earlier than the SQL procedure execution time and hence might not be optimal. Also with an SQL procedure there is also a small overhead entailed when the database manager transfers between executing procedural flow statements and SQL statements that query or modify data.

  SQL functions however are expanded and compiled within the SQL statement that references them which means that they are compiled each time that SQL statement is compiled which depending on the statement might happen dynamically. Because SQL functions are not directly associated with a package, there is no overhead entailed when the database manager transfers between executing procedural flow statements and SQL statements that query or modify data.

Choose to implement an SQL procedure if:

- SQL PL features that are only supported in SQL procedures are required. This includes: output parameter support, use of a cursor, the ability to return multiple result sets to the caller, full condition handling support, transaction and savepoint control, or other features.
- You want to execute non-SQL PL statements that can only be executed in SQL procedures.
- You want to modify data and modifying data is not supported for the type of function you need.

Although it isn't always obvious, you can often easily re-write SQL procedures as SQL functions that perform equivalent logic. This can be an effective way to maximize performance when every little performance improvement counts.

Determining when to use dynamic compound SQL statements or SQL procedures

When determining how to implement an atomic block of SQL PL and other SQL statements you might be faced with a choice between using dynamic compound SQL statements or SQL procedures. Although SQL procedures internally make use of dynamic compound SQL statements, the choice to use one or the other might depend on other factors.

Performance

If a dynamic compound SQL statement can functionally meet your needs, using one is preferable, because the SQL statements that are members of a dynamic compound SQL statement are compiled and executed as a single block. Also these statements generally perform better than CALL statements to logically equivalent SQL procedures.

At SQL procedure creation time, the procedure is compiled and a package is created. The package contains the best execution path for accessing data as of the SQL procedure compile time. Dynamic compound SQL statements are compiled when they are executed. The best execution path for accessing data for these statements is determined using the most up to date database information which can mean that their access plan can be better than that of a logically equivalent SQL procedure that was created at an earlier time which means that they might perform better.

Complexity of the required logic

If the logic to be implemented is complex, SQL procedures are likely preferable, because they support the comprehensive SQL PL with which you can easily implement efficient logic.

If the logic is quite simple and the number of statements is relatively small, consider using inline SQL PL in a dynamic compound statement or function. SQL procedures can handle simple logic, but use of SQL procedures incurs some overhead that if not required is best avoided.

Number of SQL statements to be executed

In cases where only one or two SQL statements are to be executed, there might be no benefit in using a SQL procedure. This might actually negatively impact the total performance required to execute these statements. In such a case, it is better

to use inline SQL PL in a dynamic compound statement.

Atomicity and transaction control

Atomicity is another consideration. Dynamic compound statements must be atomic. Commits and rollbacks are not supported in dynamic compound statements. If transaction control is required of if support for rollback to a savepoint is required SQL procedures must be used.

Security

Security can also be a consideration. SQL procedures can only be executed by users with EXECUTE privilege on the procedure. This can be useful if you need to limit who can execute a particular piece of logic. The ability to execute a dynamic compound statement can also be managed; however SQL procedure execution authorization provides an extra layer of security control.

Feature support

If you want to take advantage of the many features only available in SQL procedures including the use of parameters, condition and error handlers, and the ability to return one or more result sets to the caller, you must use SQL procedures.

Modularity, longevity, and re-use

SQL procedures are database objects that are persistently stored in the database and can be consistently referenced by multiple applications or scripts. Dynamic compound statements are not stored in the database and therefore the logic they contain cannot be readily re-used.

By default if SQL routines can meet your needs, use them. Generally it is a requirement to implement complex logic or to use the features supported by SQL procedures, but not available to dynamic compound statements that motivates the decision to use SQL procedures.

Rewriting SQL procedures as SQL user-defined functions

To maximize performance in a database management system, if possible, it can sometimes be beneficial to rewrite simple SQL procedures as SQL functions. Procedures and functions share the fact that their routine-bodies are implemented with a compound block that can contain SQL PL. In both, the same SQL PL statements are included within compound blocks bounded by BEGIN and END keywords.

There are some things to note when translating an SQL procedure into an SQL function:

- The primary and only reason to do this is to improve routine performance when the logic only queries data.
- In a scalar function you might have to declare variables to hold the return value to get around the fact that you cannot directly assign a value to any output parameter of the function. The output value of a user-defined scalar function is only specified in the RETURN statement for the function.
- If an SQL function is going to modify data, it must be explicitly created using the MODIFIES SQL clause so that is can contain SQL statements that modify data.

In the example that follows an SQL procedure and an SQL scalar function that are logically equivalent are shown. These two routines functionally provide the same output value given the same input values, however they are implemented and invoked in slightly different ways.

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR(20),
                           IN Pid INT,
                           OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN

  IF Vendor = 'Vendor 1'
   THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
   THEN SET price = (SELECT Price FROM V2Table
                  WHERE Pid = GetPrice.Pid);
  END IF;
END
```

This procedure takes in two input parameter values and returns an output parameter value that is conditionally determined based on the input parameter values. It uses the IF statement. This SQL procedure is invoked by executing the CALL statement. For example from the CLP, you might execute the following:

```
CALL GetPrice( 'Vendor 1',  9456, ?)
```

The SQL procedure can be rewritten as a logically-equivalent SQL table-function as follows:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
    RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
              WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

This function takes in two input parameters and returns a single scalar value, conditionally based on the input parameter values. It requires the declaration and use of a local variable named price to hold the value to be returned until the function returns whereas the SQL procedure can use the output parameter as a variable. Functionally these two routines are performing the same logic.

Now, of course the execution interface for each of these routines is different. Instead of simply calling the SQL procedure with the CALL statement, the SQL function must be invoked within an SQL statement where an expression is allowed. In most cases this isn't a problem and might actually be beneficial if the intention is to immediately operate on the data returned by the routine. Here are two examples of how the SQL function can be invoked.

It can be invoked using the VALUES statement:

```
VALUES (GetPrice('Vendor 1', 9456))
```

It can also be invoked in a SELECT statement that for example might select values from a table and filter rows based on the result of the function:

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

Chapter 4. Developing SQL procedures

Developing SQL procedures is similar to developing other types of routines. Development of SQL procedures covers all of the steps required from the design stage to the deployment stage.

- SQL procedures
- Designing SQL procedures
- Creating SQL procedures
- Calling SQL procedures
- Debugging SQL procedures
- Tuning SQL procedures
- Deploying SQL procedures

To assist you in developing SQL procedures, several examples of SQL procedures are available for reference. These examples show a variety of different kinds of SQL procedures that demonstrate some of the various SQL procedure features.

SQL procedures

SQL procedures are procedures implemented completely with SQL that can be used to encapsulate logic that can be invoked like a programming sub-routine. There are many useful applications of SQL procedures within a database or database application architecture. SQL procedures can be used to create simple scripts for quickly querying transforming, and updating data or for generating basic reports, for improving application performance, for modularizing applications, and for improving overall database design, and database security.

There are many features of SQL procedures which make them powerful routine options.

Before deciding to implement a SQL procedure, it is important that you understand what SQL procedures are in the context of SQL routines, how they are implemented, and how they can be used, by first learning about routines and then by referring to the topic, ″Overview of SQL procedures″.

Features of SQL procedures

SQL procedures are characterized by many features. SQL procedures:

- Can contain SQL Procedural Language statements and features which support the implementation of control-flow logic around traditional static and dynamic SQL statements.
- Are supported in the entire DB2 family brand of database products in which many if not all of the features supported in DB2 Version 9 are supported.
- Are easy to implement, because they use a simple high-level, strongly typed language.
- SQL procedures are more reliable than equivalent external procedures.
- Adhere to the SQL99 ANSI/ISO/IEC SQL standard.
- Support input, output, and input-output parameter passing modes.
- Support a simple, but powerful condition and error-handling model.

© Copyright IBM Corp. 1993, 2008

- Allow you to return multiple result sets to the caller or to a client application.
- Allow you to easily access the SQLSTATE and SQLCODE values as special variables.
- Reside in the database and are automatically backed up and restored.
- Can be invoked wherever the CALL statement is supported.
- Support nested procedure calls to other SQL procedures or procedures implemented in other languages.
- Support recursion.
- Support savepoints and the rolling back of executed SQL statements to provide extensive transaction control.
- Can be called from triggers.

SQL procedures provide extensive support not limited to what is listed above. When implemented according to best practices, they can play an essential role in database architecture, database application design, and in database system performance.

Designing SQL procedures

Designing SQL procedures requires an understanding of your requirements, SQL procedure features, how to use the SQL features, and knowledge of any restrictions that might impede your design. The following topics about SQL procedure design will help you learn how to design SQL procedures that make best use of SQL procedure features.

- Parts of SQL procedures
- Cross-platform SQL stored procedure considerations
- Supported SQL PL statements and language features in SQL procedures
- OLTP considerations for SQL procedures
- Performance of SQL procedures
- Rewriting SQL procedures as SQL user-defined functions
- Handling DB2 errors and warnings

Parts of SQL procedures

To understand SQL procedures, it helps to understand the parts of an SQL procedure. The following are just some of the parts of SQL procedures:

- Structure of SQL procedures
- Parameters in SQL procedures
- Variables in SQL procedures
- SQLCODE and SQLSTATE in SQL procedures
- Atomic blocks and scope of variables in SQL procedures
- Cursors in SQL procedures
- Logic elements in SQL PL
- Condition and error handlers in SQL procedures
- SQL statements that can be executed in SQL procedures

Structure of SQL procedures

SQL procedures consist of several logic parts and SQL procedure development requires you to implement these parts according to a structured format. The format is quite straight-forward and easy to follow and is intended to simplify the design and semantics of routines.

The core of an SQL procedure is a compound statement. Compound statements are bounded by the keywords BEGIN and END. These statements can be ATOMIC or NOT ATOMIC. By default they are NOT ATOMIC.

Within a compound statement, multiple optional SQL PL objects can be declared and referenced with SQL statements. The following diagram illustrates the structured format of a compound statement within SQL procedures:

```
label:  BEGIN
  Variable declarations
  Condition declarations
  Cursor declarations
  Condition handler declarations
  Assignment, flow of control, SQL statements and other compound statements
END label
```

The diagram shows that SQL procedures can consist of one or more optionally atomic compound statements (or blocks) and that these blocks can be nested or serially introduced within a single SQL procedure. Within each of these atomic blocks there is a prescribed order for the optional variable, condition, and handler declarations. These must precede the introduction of procedural logic implemented with SQL-control statements and other SQL statements and cursor declarations. Cursors can be declared anywhere with the set of SQL statements contained in the SQL procedure body.

To clarify control-flow, SQL procedure atomic blocks can be labeled as can many of the SQL control-statements contained within them. This makes it easier to be precise when referencing variables and transfer of control statement references.

Here is an example of an SQL procedure that demonstrates each of the elements listed above:

```
CREATE PROCEDURE DEL_INV_FOR_PROD (IN prod INT, OUT err_buffer VARCHAR(128))
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN

  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE integer DEFAULT 0;
  DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
  DECLARE cur1 CURSOR WITH RETURN TO CALLER
                 FOR SELECT * FROM Inv;

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

       SET err_buffer = '';

       IF (prod < 200)
         DELETE FROM Inv WHERE product = prod;
       ELSE IF (prod < 400)
         UPDATE Inv SET quantity = 0 WHERE product = prod;
       ELSE
```

```
          UPDATE Inv SET quantity = NULL WHERE product = prod;
        END IF;

    B:  OPEN cur1;

END
```

NOT ATOMIC compound statements in SQL procedures

The previous example illustrated a NOT ATOMIC compound statement and is the default type used in SQL procedures. If an unhandled error condition occurs within the compound statement, any work that is completed before the error will not be rolled back, but will not be committed either. The group of statements can only be rolled back if the unit of work is explicitly rolled back using ROLLBACK or ROLLBACK TO SAVEPOINT statements. You can also use the COMMIT statement to commit successful statements if it makes sense to do so.

Here is an example of an SQL procedure with a NOT ATOMIC compound statement:

```
CREATE PROCEDURE not_atomic_proc ()
LANGUAGE SQL
SPECIFIC not_atomic_proc
 nap:  BEGIN NOT ATOMIC

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R11:TAA', 1);

 SIGNAL SQLSTATE '70000';

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R22:TBB', 1);

END nap
```

When the SIGNAL statement is executed it explicitly raises an error that is not handled. The procedure returns immediately afterwards. After the procedure returns, although an error occurred, the first INSERT statement did successfully execute and inserted a row into the c1_sched table. The procedure neither committed, nor rolled back the row insert and this remains to be done for the complete unit of work in which the SQL procedure was called.

ATOMIC compound statements in SQL procedures

As the name suggests, ATOMIC compound statements, can be thought of as a singular whole. If any unhandled error conditions arise within it, all statements that have executed up to that point are considered to have failed as well and are therefore rolled back.

Atomic compound statements cannot be nested inside other ATOMIC compound statements.

You cannot use the SAVEPOINT statement, the COMMIT statement, or the ROLLBACK statement from within an ATOMIC compound statement. These are only supported in NOT ATOMIC compound statements within SQL procedures.

Here is an example of an SQL procedure with an ATOMIC compound statement:

```
CREATE PROCEDURE atomic_proc ()
LANGUAGE SQL
SPECIFIC atomic_proc
```

```
ap:  BEGIN ATOMIC

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R33:TCC', 1);

   SIGNAL SQLSTATE '70000';

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R44:TDD', 1);

END ap
```

When the SIGNAL statement is executed it explicitly raises an error that is not handled. The procedure returns immediately afterwards. The first INSERT statement is rolled back despite successfully executing resulting in a table with no inserted rows for this procedure.

Labels and SQL procedure compound statements

Labels can optionally be used to name any executable statement in an SQL procedure, including compound statements and loops. By referencing labels in other statements you can force the flow of execution to jump out of a compound statement or loop or additionally to jump to the beginning of a compound statement or loop. Labels can be referenced by the GOTO, ITERATE, and LEAVE statements.

Optionally you can supply a corresponding label for the END of a compound statement. If an ending label is supplied, it must be same as the label used at its beginning.

Each label must be unique within the body of an SQL procedure.

Labels can also be used to avoid ambiguity if a variable with the same name has been declared in more than one compound statement if the stored procedure. A label can be used to qualify the name of an SQL variable.

Array support in SQL procedures

SQL procedures support parameters and variables of array types. Arrays are a convenient way of passing transient collections of data between an application and a stored procedure or between two stored procedures.

Within SQL stored procedures, arrays can be manipulated as arrays in conventional programming languages. Furthermore, arrays are integrated within the relational model in such a way that data represented as an array can be easily converted into a table and data in a table column can be aggregated into an array. The examples below illustrate several operations on arrays. Both examples are command line processor (CLP) scripts that use the percentage character (%) as a statement terminator.

Example 1

This example shows two procedures, sub and main. Procedure main creates an array of 6 integers using an array constructor. It then passes the array to procedure sum, which computes the sum of all the elements in the input array and returns the result to main. Procedure sum illustrates the use of array subindexing and of the CARDINALITY function, which returns the number of elements in an array.

```
create type intArray as integer array[100] %

create procedure sum(in numList intArray, out total integer)
begin
declare i, n integer;

set n = CARDINALITY(numList);

set i = 1;
set total = 0;

while (i < n) do
set total = total + numList[i];
set i = i + 1;
end while;

end %

create procedure main(out total integer)
begin
declare numList intArray;

set numList = ARRAY[1,2,3,4,5,6];

call sum(numList, total);

end %
```

Example 2

In this example, we use two array data types (intArray and stringArray), and a persons table with two columns (id and name). Procedure processPersons adds three additional persons to the table, and returns an array with the person names that contain letter 'o', ordered by id. The ids and name of the three persons to be added are represented as two arrays (ids and names). These arrays are used as arguments to the UNNEST function, which turns the arrays into a two-column table, whose elements are then inserted into the persons table. Finally, the last set statement in the procedure uses the ARRAY_AGG aggregate function to compute the value of the output parameter.

```
create type intArray as integer array[100] %
create type stringArray as varchar(10) array[100] %

create table persons (id integer, name varchar(10)) %
insert into persons values(2, 'Tom') %
insert into persons values(4, 'Jill') %
insert into persons values(1, 'Joe') %
insert into persons values(3, 'Mary') %

create procedure processPersons(out witho stringArray)
begin
declare ids intArray;
declare names stringArray;

set ids = ARRAY[5,6,7];
set names = ARRAY['Bob', 'Ann', 'Sue'];

insert into persons(id, name)
(select T.i, T.n from UNNEST(ids, names) as T(i, n));

set witho = (select array_agg(name order by id)
from persons
where name like '%o%');
end %
```

Parameters in SQL procedures

SQL procedures support parameters for the passing of SQL values into and out of procedures.

Parameters can be useful in SQL procedures when implementing logic that is conditional on a particular input or set of input scalar values or when you need to return one or more output scalar values and you do not want to return a result set.

It is good to understand the features of and limitations of parameters in SQL procedures when designing or creating SQL procedures.

- DB2 supports the optional use of a large number of input, output, and input-output parameters in SQL procedures. The keywords IN, OUT, and INOUT in the routine signature portion of CREATE PROCEDURE statements indicate the mode or intended use of the parameter. IN and OUT parameters are passed by value, and INOUT parameters are passed by reference.
- When multiple parameters are specified for a procedure they must each have a unique name.
- If a variable is to be declared within the procedure with the same name as a parameter, the variable must be declared within a labeled atomic block nested within the procedure. Otherwise DB2 will detect what would otherwise be an ambiguous name reference.
- Parameters to SQL procedures cannot be named either of SQLSTATE or SQLCODE regardless of the data type for the parameter.

Refer to the CREATE PROCEDURE (SQL) statement for complete details about parameter references in SQL procedures.

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

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

Parameter markers

A parameter marker, often denoted by a question mark (?), is a place holder in an SQL statement whose value is obtained during statement execution. An application associates parameter markers to application variables. During the execution of the statement, the values of these variables replace each respective parameter marker. Data conversion might take place during the process.

Benefits of parameter markers

For SQL statements that need to be executed many times, it is often beneficial to prepare the SQL statement once, and reuse the query plan by using parameter markers to substitute the input values during runtime. In DB2® 9, a parameter marker is represented in one of two ways:

- The first style, with a "?" character, is used in dynamic SQL execution (dynamic Embedded SQL, CLI, Perl, etc).

- The second style represents the embedded SQL standard construction where the name of the variable is prefixed with a colon (:var1) . This style is used in static SQL execution and is commonly referred to as a host variable.

Use of either style indicates where an application variable is to be substituted inside an SQL statement. Parameter markers are referenced by number, and are numbered sequentially from left to right, starting at one. Before the SQL statement is executed, the application must bind a variable storage area to each parameter marker specified in the SQL statement. In addition, the bound variables must be a valid storage area, and must contain input data values when the prepared statement is executed against the database.

The following example illustrates an SQL statement containing two parameter markers.

```
SELECT * FROM customers WHERE custid = ? AND lastname = ?
```

Supported types

DB2 supports untyped parameter markers, which can be used in selected locations of an SQL statement. Table 1 lists the restrictions on parameter marker usage.

Table 8. Restrictions on parameter marker usage

Untyped parameter marker location	Data type
Expression: Alone in a select list	Error
Expression: Both operands of an arithmetic operator	Error
Predicate: Left-hand side operand of an IN predicate	Error
Predicate: Both operands of a relational operator	Error
Function: Operand of an aggregation function	Error

Examples

DB2® provides a rich set of standard interfaces including CLI/ODBC, JDBC, and ADO.NET to access data efficiently. The following code snippets show the use of prepared statement with parameter markers for each data access API.

Consider the following table schema for table t1, where column c1 is the primary key for table t1.

Table 9. Example table schema

Column name	DB2 data type	Nullable
c1	INTEGER	false
c2	SMALLINT	true
c3	CHAR(20)	true
c4	VARCHAR(20)	true
c5	DECIMAL(8,2)	true
c6	DATE	true

Table 9. Example table schema (continued)

Column name	DB2 data type	Nullable
c7	TIME	true
c8	TIMESTAMP	true
c9	BLOB(30)	true

The following examples illustrate how to insert a row into table t1 using a prepared statement.

CLI Example

```
void parameterExample1(void)
{
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;
   SQLRETURN rc;
   TCHAR server[] = _T("C:\\mysample\\");
   TCHAR uid[] = _T("db2e");
   TCHAR pwd[] = _T("db2e");
   long p1 = 10;
   short p2 = 100;
   TCHAR p3[100];
   TCHAR p4[100];
   TCHAR p5[100];
   TCHAR p6[100];
   TCHAR p7[100];
   TCHAR p8[100];
   char  p9[100];
   long len = 0;

   _tcscpy(p3, _T("data1"));
   _tcscpy(p4, _T("data2"));
   _tcscpy(p5, _T("10.12"));
   _tcscpy(p6, _T("2003-06-30"));
   _tcscpy(p7, _T("12:12:12"));
   _tcscpy(p8, _T("2003-06-30-17.54.27.710000"));

   memset(p9, 0, sizeof(p9));
   p9[0] = 'X';
   p9[1] = 'Y';
   p9[2] = 'Z';

   rc = SQLAllocEnv(&henv);
   // check return code ...

   rc = SQLAllocConnect(henv, &hdbc);
   // check return code ...

   rc = SQLConnect(hdbc, (SQLTCHAR*)server, SQL_NTS,
    (SQLTCHAR*)uid, SQL_NTS, (SQLTCHAR*)pwd, SQL_NTS);
   // check return code ...

   rc = SQLAllocStmt(hdbc, &hstmt);
   // check return code ...

   // prepare the statement
   rc = SQLPrepare(hstmt, _T("INSERT INTO t1 VALUES (?,?,?,?,?,?,?,?,?)"), SQL_NTS);
   // check return code ...

   // bind input parameters
   rc = SQLBindParameter(hstmt, (unsigned short)1, SQL_PARAM_INPUT,
    SQL_C_LONG, SQL_INTEGER, 4, 0, &p1, sizeof(p1), &len);
```

```
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)2, SQL_PARAM_INPUT, SQL_C_LONG,
  SQL_SMALLINT, 2, 0, &p2, sizeof(p2), &len);
// check return code ...

len = SQL_NTS;
rc = SQLBindParameter(hstmt, (unsigned short)3, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_CHAR, 0, 0, &p3[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)4, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_VARCHAR, 0, 0, &p4[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)5, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_DECIMAL, 8, 2, &p5[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)6, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_DATE, 0, 0, &p6[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)7, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIME, 0, 0, &p7[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)8, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIMESTAMP, 0, 0, &p8[0], 100, &len);
// check return code ...

len = 3;
rc = SQLBindParameter(hstmt, (unsigned short)9, SQL_PARAM_INPUT, SQL_C_BINARY,
  SQL_BINARY, 0, 0, &p9[0], 100, &len);
// check return code ...

// execute the prepared statement
rc = SQLExecute(hstmt);
// check return code ...

rc = SQLFreeStmt(hstmt, SQL_DROP);
// check return code ...

rc = SQLDisconnect(hdbc);
// check return code ...

rc = SQLFreeConnect(hdbc);
// check return code ...

rc = SQLFreeEnv(henv);
// check return code ...
```

C Example

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;

/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

JDBC Example

```
public static void parameterExample1() {

   String driver = "com.ibm.db2e.jdbc.DB2eDriver";
   String url    = "jdbc:db2e:mysample";
   Connection conn = null;
   PreparedStatement pstmt = null;

   try
   {
      Class.forName(driver);

      conn = DriverManager.getConnection(url);

      // prepare the statement
      pstmt = conn.prepareStatement("INSERT INTO t1 VALUES
                                                    (?, ?, ?, ?, ?, ?, ?, ?, ?)");

      // bind the input parameters
      pstmt.setInt(1, 1);
      pstmt.setShort(2, (short)2);
      pstmt.setString(3, "data1");
      pstmt.setString(4, "data2");
      pstmt.setBigDecimal(5, new java.math.BigDecimal("12.34"));
      pstmt.setDate(6, new java.sql.Date(System.currentTimeMillis() ) );
      pstmt.setTime(7, new java.sql.Time(System.currentTimeMillis() ) );
      pstmt.setTimestamp (8, new java.sql.Timestamp(System.currentTimeMillis() ) );
      pstmt.setBytes(9, new byte[] { (byte)'X', (byte)'Y', (byte)'Z' } );

      // execute the statement
      pstmt.execute();

      pstmt.close();

      conn.close();
   }
   catch (SQLException sqlEx)
   {
      while(sqlEx != null)
      {
          System.out.println("SQLERROR: \n" + sqlEx.getErrorCode() +
             ", SQLState: " + sqlEx.getSQLState() +
             ", Message: " + sqlEx.getMessage() +
             ", Vendor: " + sqlEx.getErrorCode() );
         sqlEx = sqlEx.getNextException();
      }
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
   }
}
```

ADO.NET Example [C#]

```
public static void ParameterExample1()
{
   DB2eConnection conn = null;
   DB2eCommand cmd  = null;
   String connString   = @"database=.\; uid=db2e; pwd=db2e";
   int i = 1;

   try
   {
      conn = new DB2eConnection(connString);

      conn.Open();
```

```
   cmd = new DB2eCommand("INSERT INTO t1 VALUES
                                    (?, ?, ?, ?, ?, ?, ?, ?, ?)", conn);

   // prepare the command
   cmd.Prepare();

   // bind the input parameters
   DB2eParameter p1 = new DB2eParameter("@p1", DB2eType.Integer);
   p1.Value = ++i;
   cmd.Parameters.Add(p1);

   DB2eParameter p2 = new DB2eParameter("@p2", DB2eType.SmallInt);
   p2.Value = 100;
   cmd.Parameters.Add(p2);

   DB2eParameter p3 = new DB2eParameter("@p3", DB2eType.Char);
   p3.Value = "data1";
   cmd.Parameters.Add(p3);

   DB2eParameter p4 = new DB2eParameter("@p4", DB2eType.VarChar);
   p4.Value = "data2";
   cmd.Parameters.Add(p4);

   DB2eParameter p5 = new DB2eParameter("@p5", DB2eType.Decimal);
   p5.Value = 20.25;
   cmd.Parameters.Add(p5);

   DB2eParameter p6 = new DB2eParameter("@p6", DB2eType.Date);
   p6.Value = DateTime.Now;
   cmd.Parameters.Add(p6);

   DB2eParameter p7 = new DB2eParameter("@p7", DB2eType.Time);
   p7.Value = new TimeSpan(23, 23, 23);
   cmd.Parameters.Add(p7);

   DB2eParameter p8 = new DB2eParameter("@p8", DB2eType.Timestamp);
   p8.Value = DateTime.Now;
   cmd.Parameters.Add(p8);

   byte []barr = new byte[3];
   barr[0] = (byte)'X';
   barr[1] = (byte)'Y';
   barr[2] = (byte)'Z';

   DB2eParameter p9 = new DB2eParameter("@p9", DB2eType.Blob);
   p9.Value = barr;
   cmd.Parameters.Add(p9);

   // execute the prepared command
   cmd.ExecuteNonQuery();
}
catch (DB2eException e1)
{
   for (int i=0; i < e1.Errors.Count; i++)
   {
      Console.WriteLine("Error #" + i + "\n" +
         "Message: " + e1.Errors[i].Message + "\n" +
         "Native: " + e1.Errors[i].NativeError.ToString() + "\n" +
         "SQL: " + e1.Errors[i].SQLState + "\n");
   }
}
catch (Exception e2)
{
    Console.WriteLine(e2.Message);
}
finally
```

```
   {
      if (conn != null && conn.State != ConnectionState.Closed)
      {
            conn.Close();
            conn = null;
      }
   }
}
```

Variables in SQL procedures (DECLARE, SET statements)

Local variable support in SQL procedures allows you to assign and retrieve SQL values in support of SQL procedure logic.

Variables in SQL procedures are defined by using the DECLARE statement.

Values can be assigned to variables using the SET statement or the SELECT INTO statement or as a default value when the variable is declared. Literals, expressions, the result of a query, and special register values can be assigned to variables.

Variable values can be assigned to SQL procedure parameters, other variables in the SQL procedure, and can be referenced as parameters within SQL statements that executed within the routine.

The following example demonstrates various methods for assigning and retrieving variable values.

```
CREATE PROCEDURE proc_vars()
SPECIFIC proc_vars
LANGUAGE SQL
BEGIN

  DECLARE v_rcount INTEGER;

  DECLARE v_max DECIMAL (9,2);

  DECLARE v_adate, v_another  DATE;

  DECLARE v_total INTEGER DEFAULT 0;          -- (1)

  SET v_total = v_total + 1                   -- (2)

  SELECT MAX(salary)                          -- (3)
    INTO v_max FROM employee;

  VALUES CURRENT_DATE INTO v_date;            -- (4)

  SELECT CURRENT DATE, CURRENT DATE           -- (5)
       INTO v_adate, v_another
  FROM SYSIBM.SYSDUMMY1;

  DELETE FROM T;
  GET DIAGNOSTICS v_rcount = ROW_COUNT;       -- (6)

END
```

When declaring a variable, you can specify a default value using the DEFAULT clause as in line (1). Line (2) shows that a SET statement can be used to assign a single variable value. Variables can also be set by executing a SELECT or FETCH statement in combination with the INTO clause as shown in line (3). Lines (4) and (5) show how the VALUES INTO statement can be used to evaluate a function or special register and assign the value to a variable or to multiple variables.

You can also assign the result of a GET DIAGNOSTICS statement to a variable. GET DIAGNOSTICS can be used to get a handle on the # of affected rows (updated for an UPDATE statement, DELETE for a DELETE statement) or to get the return status of a just executed SQL statement. Line (6) shows how the number of rows modified by the just previously executed DELETE statement can be assigned to a variable.

XML and XQuery support in SQL procedures

SQL procedures support parameters and variables of data type XML. They can be used in SQL statements in the same way as variables of any other data type. In addition, variables of data type XML can be passed as parameters to XQuery expressions in XMLEXISTS, XMLQUERY and XMLTABLE expressions.

The following example shows the declaration, use, and assignment of XML parameters and variables in an SQL procedure:

```
CREATE TABLE T1(C1 XML) %

CREATE PROCEDURE proc1(IN parm1 XML, IN parm2 VARCHAR(32000))
LANGUAGE SQL
BEGIN
   DECLARE var1 XML;

   /* check if the value of XML parameter parm1
      contains an item with a value less than 200 */
   IF(XMLEXISTS('$x/ITEM[value < 200]' passing by ref parm1 as "x"))THEN

     /* if it does, insert the value of parm1 into table T1 */
     INSERT INTO T1 VALUES(parm1);

   END IF;

   /* parse parameter parm2's value and assign it to a variable */
   SET var1 = XMLPARSE(document parm2 preserve whitespace);

   /* insert variable var1 into table T1
   INSERT INTO T1 VALUES(var1);

END %
```

In the example above there is a table `T1` with an XML column. The SQL procedure accepts two parameters of data type XML named `parm1` and `parm2`. Within the SQL procedure an XML variable is declared named `var1`.

The logic of the SQL procedure checks if the value of XML parameter `parm1` contains an item with a value less than 200. If it does, the XML value is directly inserted into column `C1` in table `T1`.

Then the value of parameter `parm2` is parsed using the XMLPARSE function and assigned to XML variable `var1`. The XML variable value is then also inserted into column `C1` in table `T1`.

The ability to implement control flow logic around XQuery operations makes it easy to develop complex algorithms that query and access XML data stored in a database.

SQLCODE and SQLSTATE variables in SQL procedures

To perform error handling or to help you debug your SQL procedures, you might find it useful to test the value of the SQLCODE or SQLSTATE values, return these

values as output parameters or as part of a diagnostic message string, or insert these values into a table to provide basic tracing support.

To use the SQLCODE and SQLSTATE values within SQL procedures, you must declare the following SQL variables in the SQL procedure body:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

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

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

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

  executable-statements
END
```

Note: When you access the SQLCODE or SQLSTATE variables in an SQL procedure, DB2 sets the value of SQLCODE to 0 and SQLSTATE to '00000' for the subsequent statement.

Compound statements and scope of variables in SQL procedures

SQL procedures can contain one or more compound statements. They can be introduced in serial or can be nested within another compound statement. Each compound statement introduces a new scope in which variables might or might not be available for use.

The use of labels to identify a compound statement is important as the label can be used to qualify and uniquely identify variables declared within the compound statement. This is particularly important when referencing of variables in different compound statements or in nested compound statements.

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

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

```
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

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

    lab2: END;
END lab1@
```

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

Effect of commits and rollbacks on XML parameter and variable values in SQL procedures

Commits and rollbacks within SQL procedures affect the values of parameters and variables of data type XML. During the execution of SQL procedures, upon a commit or rollback operation, the values assigned to XML parameters and XML variables will no longer be available.

Attempts to reference an SQL variable or SQL parameter of data type XML after a commit or rollback operation will cause an error (SQL1354N, 560CE) to be raised.

To successfully reference XML parameters and variables after a commit or rollback operation occurs, new values must first be assigned to them.

Consider the availability of XML parameter and variable values when adding ROLLBACK and COMMIT statements to SQL procedures.

Cursors in SQL procedures

In SQL procedures, a cursor make it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, an SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

A cursor can be viewed as a pointer to one row in a set of rows. The cursor can only reference one row at a time, but can move to other rows of the result set as needed.

To use cursors in SQL procedures, you need to do the following:

1. Declare a cursor that defines a result set.
2. Open the cursor to establish the result set.
3. Fetch the data into local variables as needed from the cursor, one row at a time.
4. Close the cursor when done

To work with cursors you must use the following SQL statements:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

The following example demonstrates the basic use of a read-only cursor within an SQL procedure:

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE p_sum INTEGER;
  DECLARE p_sal INTEGER;
  DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

   SET p_sum = 0;

   OPEN c;

   FETCH FROM c INTO p_sal;

   WHILE(SQLSTATE = '00000') DO
      SET p_sum = p_sum + p_sal;
      FETCH FROM c INTO p_sal;
   END WHILE;

   CLOSE c;

   SET sum = p_sum;

END%
```

Here is a more complex example of use of a cursor within an SQL procedure. This example demonstrates the combined use of a cursor and SQL PL statements.

Cursors for XQuery expressions in SQL procedures

SQL Procedures support the definition of cursors on XQuery expressions. A cursor on an XQuery expression allows you to iterate over the elements of the XQuery sequence returned by the expression.

Unlike cursors defined on SQL statements, which can be defined either statically or dynamically, cursors on XQuery expressions can only be defined dynamically. To declare a cursor dynamically, it is necessary to declare a variable of type CHAR or VARCHAR to contain the XQuery expression that will define the cursor result set. The XQuery expression must be prepared before the cursor can be opened and the result set resolved.

An example of an SQL procedure that dynamically declares a cursor for an XQuery expression, opens the cursor, and fetches XML data is shown here:

```
CREATE PROCEDURE xmlProc(IN inCust XML, OUT resXML XML)
SPECIFIC xmlProc
LANGUAGE SQL
BEGIN
    DECLARE SQLSTATE CHAR(5);
    DECLARE stmt_text VARCHAR (1024);
    DECLARE customer XML;
    DECLARE cityXml XML;
    DECLARE city VARCHAR (100);
    DECLARE stmt STATEMENT;
    DECLARE cur1 CURSOR FOR stmt;

   -- Get the city of the input customer
   SET cityXml = XMLQUERY('$cust/customerinfo//city' passing inCust as "cust");
   SET city = XMLCAST(cityXml as VARCHAR(100));

  -- Iterate over all the customers from the city using an XQUERY cursor
  -- and collect the customer name values into the output XML value
```

```
   SET stmt_text = 'XQUERY for $cust
                    in db2-fn:xmlcolumn("CUSTOMER.INFO")
                       /*:customerinfo/*:addr[*:city= "' || city ||'"]
                    return <Customer>{$cust/../@Cid}{$cust/../*:name}</Customer>';

  -- Use the name of the city for the input customer data as a prefix
  SET resXML = cityXml;

  PREPARE stmt FROM stmt_text;
  OPEN cur1;

  FETCH cur1 INTO customer;
  WHILE (SQLSTATE = '00000') DO
    SET resXML = XMLCONCAT(resXML, customer);
    FETCH cur1 INTO customer;
  END WHILE;

  set resXML = XMLQUERY('<result> {$res} </result>'
                        passing resXML as "res");

END
```

This SQL procedure collects the IDs and names of customers defined in a table name CUSTOMER that are located in the same city as the customer for which XML data is provided as an input parameter.

The SQL procedure above can be called by executing the CALL statement as follows:

```
  CALL xmlProc(xmlparse(document '<customerinfo Cid="5002">
                                    <name>Jim Noodle</name>
                                    <addr country="Canada">
                                       <street>25 EastCreek</street>
                                       <city>Markham</city>
                                       <prov-state>Ontario</prov-state>
                                       <pcode-zip>N9C-3T6</pcode-zip>
                                    </addr>
                                    <phone type="work">905-566-7258</phone>
                                 </customerinfo>' PRESERVE WHITESPACE),?,?)
```

If this SQL procedure is created and run against the SAMPLE database, it returns XML data for two customers.

Since parameter markers are not supported for XML values, a workaround to this limitation is to construct a dynamic SQL statement out of concatenated statement fragments that include the value of one or more local variables.

For example:

```
DECLARE person_name VARCHAR(128);

SET person_name = "Joe";
SET stmt_text = ' for $fname in db2-fn:sqlquery
   ("SELECT doc
      FROM T1
      WHERE DOCID=1")//fullname where $fname/first = ''' person_name || ''';
```

This example returns a result set in a variable assignment for an XQuery statement that contains an SQL fullselect. The result set contains the full names of persons with the first name `Joe`. Functionally, the SQL portion selects the XML documents from column `doc` in table `T1` that have an ID of 1. The XQuery portion then selects the fullname values in the XML documents where the value first is `Joe`.

SQL PL logic elements in the SQL-procedure body

Sequential execution is the most basic path that program execution can take. With this method, the program starts execution at the first line of the code, followed by the next, and continues until the final statement in the code has been executed. This approach works fine for very simple tasks, but tends to lack usefulness because it can only handle one situation. Programs often need to be able to decide what to do in response to changing circumstances. By controlling a code's execution path, a specific piece of code can then be used to intelligently handle more than one situation.

SQL PL provides support for variables and flow of control statements that can be used to control the sequence of statement execution. Statements such as IF and CASE are used to conditionally execute blocks of SQL PL statements, while other statements, such as WHILE and REPEAT, are typically used to execute a set of statements repetitively until a task is complete.

Although there are many types of SQL PL statements, there are a few categories into which these can be sorted:

- Variable related statements
- Conditional statements
- Loop statements
- Transfer of control statements

Variable related statements in SQL procedures

Variable related SQL statements are used to declare variables and to assign values to variables. There are a few types of variable related statements:

- DECLARE <variable> statement in SQL procedures
- DECLARE <condition> statement in SQL procedures
- DECLARE <condition handler> statement in SQL procedures
- DECLARE CURSOR in SQL procedures
- SET (assignment-statement) in SQL procedures

These statements provide the necessary support required to make use of the other types of SQL PL statements and SQL statements that will make use of variable values.

Conditional statements in SQL procedures

Conditional statements are used to define what logic is to be executed based on the status of some condition being satisfied. There are two types of conditional statements supported in SQL procedures:

- CASE
- IF

These statements are similar; however the CASE statements extends the IF statement.

CASE statement in SQL procedures

CASE statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. There are two types of CASE statements:

- Simple case statement: used to enter into some logic based on a literal value

- Searched case statement: used to enter into some logic based on the value of an expression

The WHEN clause of the CASE statement defines the value that when satisfied determines the flow of control.

Here is an example of an SQL procedure with a CASE statement with a simple-case-statement-when-clause:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE  v_workdept
    WHEN 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
    END CASE

END
```

Here is an example of CASE statement with a searched-case-statement-when-clause:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE
    WHEN v_workdept = 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN v_workdept = 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
  END CASE

END
```

The examples provided above are logically equivalent, however it is important to note that CASE statements with a searched-case-statement-when-clause can be very powerful. Any supported SQL expression can be used here. These expressions can contain references to variables, parameters, special registers, and more.

IF statement in SQL procedures

IF statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. The IF statement is logically equivalent to a CASE statements with a searched-case-statement-when clause.

The IF statement supports the use of optional ELSE IF clauses and a default ELSE clause. An END IF clause is required to indicate the end of the statement.

Here is an example of procedure that contains an IF statement:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
                             INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
    UPDATE employee
    SET salary = salary * 1.10, bonus = 1000
      WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
    SET salary = salary * 1.05, bonus = 500
      WHERE empno = empNum;
  ELSE
    UPDATE employee
    SET salary = salary * 1.03, bonus = 0
      WHERE empno = empNum;
  END IF;
END
```

Looping statements in SQL procedures

Looping statements provide support for repeatedly executing some logic until a condition is met. The following looping statements are supported in SQL PL:

- FOR
- LOOP
- REPEAT
- WHILE

The FOR statement is distinct from the others, because it is used to iterate over rows of a defined result set, whereas the others are using for iterating over a series of SQL statements until for each a condition is satisfied.

Labels can be defined for all loop-control-statements to identify them.

FOR statement in SQL procedures

FOR statements are a special type of looping statement, because they are used to iterate over rows in a defined read-only result set. When a FOR statement is executed a cursor is implicitly declared such that for each iteration of the FOR-loop the next row is the result set if fetched. Looping continues until there are no rows left in the result set.

The FOR statement simplifies the implementation of a cursor and makes it easy to retrieve a set of column values for a set of rows upon which logical operations can be performed.

Here is an example of an SQL procedure that contains only a simple FOR statement:

```
CREATE PROCEDURE P()
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE fullname CHAR(40);

  FOR v AS cur1 CURSOR FOR
             SELECT firstnme, midinit, lastname FROM employee
  DO
    SET fullname = v.lastname || ',' || v.firstnme
                   ||' ' || v.midinit;
    INSERT INTO tnames VALUES (fullname);
  END FOR;
END
```

Note: Logic such as is shown in the example above would be better implemented using the CONCAT function. The simple example serves to demonstrate the syntax.

The for-loop-name specifies a label for the implicit compound statement generated to implemented the FOR statement. It follows the rules for the label of a compound statement. The for-loop-name can be used to qualify the column names in the result set as returned by the select-statement.

The cursor-name simply names the cursor that is used to select the rows from the result set. If it is not specified, the DB2 database manager will automatically generate a unique cursor name internally.

The column names of the select statement must be unique and a FROM clause specifying a table (or multiple tables if doing some kind of JOIN or UNION) is required. The tables and columns referenced must exist prior to the loop being executed. Global temporary tables can be referenced.

Positioned updates and deletes, and searched updates and deletes are supported in the FOR loop.

The cursor that is created in support of the FOR statement cannot be referenced outside of the FOR loop.

LOOP statement in SQL procedures

The LOOP statement is a special type of looping statement, because has no terminating condition clause. It defines a series of statements that are executed repeatedly until another piece of logic, generally a transfer of control statement, forces the flow of control to jump to some point outside of the loop.

The LOOP statement is generally used in conjunction with one of the following statements: LEAVE, GOTO, ITERATE, or RETURN. These statements can force control to just after the loop, to a specified location in the SQL procedure, to the start of the loop to begin another iteration of the loop, or to exit the SQL procedure. To indicate where to pass flow to when using these statements, labels are used.

The LOOP statement is useful when you have complicated logic in a loop which you might need to exit in more than one way, however it should be used with care to avoid instances of infinite loops.

If the LOOP statement is used alone without a transfer of control statement, the series of statements included in the loop will be executed indefinitely or until a database condition occurs that raises a condition handler that forces a change in the control flow or a condition occurs that is not handled that forces the return of the SQL procedure.

Here is an example of an SQL procedure that contains a LOOP statement. It also uses the ITERATE and LEAVE statements.

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
```

```
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP

    FETCH c1 INTO v_deptno, v_deptname;

    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END  IF;

    INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;
END
```

WHILE statement in SQL procedures

The WHILE statement defines a set of statements to be executed until a condition that is evaluated at the beginning of the WHILE loop is false. The while-loop-condition (an expression) is evaluated before each iteration of the loop.

Here is an example of an SQL procedure with a simple WHILE loop:

```
CREATE PROCEDURE sum_mn (IN p_start INT
                        ,IN p_end INT
                        ,OUT p_sum INT)
SPECIFIC sum_mn
LANGUAGE SQL
smn: BEGIN

DECLARE v_temp INTEGER DEFAULT 0;
DECLARE v_current INTEGER;

SET v_current = p_start;

WHILE (v_current <= p_end) DO
  SET v_temp = v_temp + v_current;
  SET v_current = v_current + 1;
END WHILE;
p_sum = v_current;
END smn;
```

Note: Logic such as is shown in the example above would be better implemented using a mathematical formula. The simple example serves to demonstrate the syntax.

REPEAT statement in SQL procedures

The REPEAT statement defines a set of statements to be executed until a condition that is evaluated at the end of the REPEAT loop is true. The repeat-loop-condition is evaluated at the completion of each iteration of the loop.

With a WHILE statement, the loop is not entered if the while-loop-condition is false at 1st pass. The REPEAT statement is useful alternative; however it is noteworthy that while-loop logic can be rewritten as a REPEAT statement.

Here is an SQL procedure that includes a REPEAT statement:

```
CREATE PROCEDURE sum_mn2 (IN p_start INT
                         ,IN p_end INT
                         ,OUT p_sum INT)
SPECIFIC sum_mn2
LANGUAGE SQL
smn2: BEGIN

  DECLARE v_temp INTEGER DEFAULT 0;
  DECLARE v_current INTEGER;

  SET v_current = p_start;

  REPEAT
    SET v_temp = v_temp + v_current;
    SET v_current = v_current + 1;
  UNTIL (v_current > p_end)
  END REPEAT;
END
```

Transfer of control statements in SQL procedures

Transfer of control statements are used to redirect the flow of control within an SQL procedure. This unconditional branching can be used to cause the flow of control to jump from one point to another point, which can either precede or follow the transfer of control statement. The supported transfer of control statements in SQL procedures are:

- GOTO
- ITERATE
- LEAVE
- RETURN

Transfer of control statements can be used anywhere within an SQL procedure, however ITERATE and LEAVE are generally used in conjunction with a LOOP statement or other looping statements.

GOTO statement in SQL procedures

The GOTO statement is a straightforward and basic flow of control statement that causes an unconditional change in the flow of control. It is used to branch to a specific user-defined location using labels defined in the SQL procedure.

Use of the GOTO statement is generally considered to be poor programming practice and is not recommended. Extensive use of GOTO tends to lead to unreadable code especially when procedures grow long. Besides, GOTO is not necessary because there are better statements available to control the execution path. There are no specific situations that require the use of GOTO; instead it is more often used for convenience.

Here is an example of an SQL procedure that contains a GOTO statement:

```
CREATE PROCEDURE adjust_salary ( IN p_empno CHAR(6),
                      IN p_rating INTEGER,
     OUT p_adjusted_salary DECIMAL (8,2) )
LANGUAGE SQL
BEGIN
  DECLARE new_salary DECIMAL (9,2);
  DECLARE service DATE;  -- start date

  SELECT salary, hiredate INTO v_new_salary, v_service
    FROM employee
       WHERE empno = p_empno;
```

```
  IF service > (CURRENT DATE - 1 year) THEN
    GOTO exit;
  END IF;

  IF p_rating = 1 THEN
    SET new_salary = new_salary + (new_salary * .10);
  END IF;

  UPDATE employee SET salary = new_salary WHERE empno = p_empno;

exit:
  SET p_adjusted_salary = v_new_salary;


END
```

This example demonstrates what of the good uses of the GOTO statement: skipping almost to the end of a procedure or loop so as not to execute some logic, but to ensure that some other logic does still get executed.

You should be aware of a few additional scope considerations when using the GOTO statement:

- If the GOTO statement is defined in a FOR statement, the label must be defined inside the same FOR statement, unless it is in a nested FOR statement or nested compound statement.
- If the GOTO statement is defined in a compound statement, the label must be defined in side the same compound statement, unless it is in a nested FOR statement or nested compound statement.
- If the GOTO statement is defined in a handler, the label must be defined in the same handler, following the other scope rules.
- If the GOTO statement is defined outside of a handler, the label must not be defined within a handler.
- If the label is not defined within a scope that the GOTO statement can reach, an error is returned (SQLSTATE 42736).

ITERATE statement in SQL procedures

The ITERATE statement is used to cause the flow of control to return to the beginning of a labeled LOOP statement.

Here is an example of an SQL procedure that contains an ITERATE statement:

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP
   FETCH c1 INTO v_deptno, v_deptname;
      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END  IF;
```

```
        INSERT  INTO department (deptno, deptname)
        VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;

  END
```

In the example, the ITERATE statement is used to return the flow of control to the LOOP statement defined with label ins_loop when a column value in a fetched row matches a certain value. The position of the ITERATE statement ensures that no values are inserted into the department table.

LEAVE statement in SQL procedures

The LEAVE statement is used to transfer the flow of control out of a loop or compound statement.

Here is an example of an SQL procedure that contain a LEAVE statement:

```
  CREATE  PROCEDURE ITERATOR()
  LANGUAGE SQL
  BEGIN
    DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';

    DECLARE c1 CURSOR FOR SELECT deptno, deptname
                          FROM department ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;

    OPEN c1;

    ins_loop: LOOP

      FETCH c1 INTO v_deptno, v_deptname;

      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;
END
```

In the example, the LEAVE statement is used to exit the LOOP statement defined with label ins_loop. It is nested within an IF statement and therefore is conditionally executed when the IF-condition is true which becomes true when there are no more rows found in the cursor. The position of the LEAVE statement ensures that no further iterations of the loop are executed once a NOT FOUND error is raised.

RETURN statement in SQL procedures

The RETURN statement is used to unconditionally and immediately terminate an SQL procedure by returning the flow of control to the caller of the stored procedure.

It is mandatory that when the RETURN statement is executed that it return an integer value. If the return value is not provided, the default is 0. The value is typically used to indicate success or failure of the procedure's execution. The value can be a literal, variable, or an expression that evaluates to an integer value.

You can use one or more RETURN statements in a stored procedure. The RETURN statement can be used anywhere after the declaration blocks within the SQL-procedure-body.

To return multiple output values, parameters can be used instead. Parameter values must be set prior to the RETURN statement being executed.

Here is an example of an SQL procedure that uses the RETURN statement:

```
CREATE PROCEDURE return_test (IN p_empno CHAR(6),
                              IN p_emplastname VARCHAR(15) )
LANGUAGE SQL
SPECIFIC return_test
BEGIN

  DECLARE v_lastname VARCHAR (15);

  SELECT lastname INTO v_lastname
    FROM employee
  WHERE empno = p_empno;

  IF v_lastname = p_emplastname THEN
    RETURN 1;
  ELSE
    RETURN -1;
  END IF;

END rt
```

In the example, if the parameter *p_emplastname* matches the value stored in table employee, the procedure returns 1. If it does not match, it returns -1.

Condition handlers in SQL procedures

Condition handlers determine the behavior of your SQL procedure when a condition occurs. You can declare one or more condition handlers in your SQL procedure for general conditions, named conditions, or specific SQLSTATE values.

If a statement in your SQL procedure raises an SQLWARNING or NOT FOUND condition, and you have declared a handler for the respective condition, DB2 passes control to the corresponding handler. If you have not declared a handler for such a condition, DB2 passes control to the next statement in the SQL procedure body. If the SQLCODE and SQLSTATE variables have been declared, they will contain the corresponding values for the condition.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 passes control to that handler. If the SQLSTATE and SQLCODE variables have been declared, their values after the successful execution of a handler will be '00000' and 0 respectively.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you have not declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 terminates the SQL procedure and returns to the caller.

Returning result sets from SQL procedures

In SQL procedures, cursors can be used to do more than iterate through rows of a result set. They can also be used to return result sets to the calling program. Result sets can be retrieved by SQL procedures (in the case of a nested procedure calls) or client applications programmed in C using the CLI application programming interface, Java, CLI, or .NET CLR languages.

Prerequisites

- Authority to create an SQL procedure

To return a result set from an SQL procedure, you must:

1. Specify the DYNAMIC RESULT SETS clause in the CREATE PROCEDURE statement
2. DECLARE the cursor using the WITH RETURN clause
3. Open the cursor in the SQL procedure
4. Keep the cursor open for the client application - do not close it

Here is an example of an SQL procedure that only returns a single result set:

```
CREATE PROCEDURE  read_emp()
SPECIFIC read_emp
LANGUAGE SQL
DYNAMIC RESULT SETS 1

Re:  BEGIN

  DECLARE c_emp CURSOR WITH RETURN FOR
    SELECT salary, bonus, comm.
    FROM employee
    WHERE job != 'PRES';

  OPEN c_emp;

END Re
```

If the cursor is closed using the CLOSE statement prior to the return of the SQL procedure, the cursor result set will not be returned to the caller or client application.

Multiple result sets can be returned from an SQL procedure by using multiple cursors. To return multiple cursors the following must be done:

- Specify the DYNAMIC RESULT SETS clause in the CREATE PROCEDURE statement. Specify the maximum possible number of result sets likely to be returned. The number of results sets actually returned must not exceed this number.
- Declare cursors for each of the result sets to be returned that specify the WITH RETURN clause.
- Open the cursors to be returned.
- Keep the cursor open for the client application - do not close them.

One cursor is required per result set that is to be returned.

Result sets are returned to the caller in the order in which they are opened.

Once you have created the SQL procedure that returns a result set you might want to call it and retrieve the result set.

Receiving procedure result sets in SQL routines

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

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

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

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

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
```

2. Invoke the procedure. For example:

```
CALL targetProcedure();
```

3. ASSOCIATE the result set locator variables (defined above) with the invoked procedure. For example:

```
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
```

4. ALLOCATE the result set cursors passed from the invoked procedure to the result set locators. For example:

```
ALLOCATE rsCur CURSOR FOR RESULT SET result1;
```

5. FETCH rows from the result sets. For example:

```
FETCH rsCur INTO ...
```

Creating SQL procedures

Creating SQL procedures is similar to creating any database object in that it consists of executing a DDL SQL statement.

SQL procedures are created by executing the CREATE PROCEDURE statement which can be done using graphical development environment tools or by directly executing the statement from the DB2 Command Line Processor (CLP), a DB2 Command Window, the DB2 Command Editor, or another DB2 interface.

When creating SQL procedures, you can specify how the precompiler and binder should generate the procedure package, what authorization ID should be used to set the SQL procedure definer in the DB2 catalog views, and to set other package options.

Creating SQL procedures from the command line

Prerequisites

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

Procedure

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

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

  For example, in the following CREATE PROCEDURE statement, the 'at;' sign ('@') is used as the terminating character for a DB2 CLP script named myCLPscript.db2:

```
    CREATE PROCEDURE UPDATE_SALARY_IF
    (IN employee_number CHAR(6), IN rating SMALLINT)
    LANGUAGE SQL
    BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

      IF (rating = 1)
        THEN UPDATE employee
          SET salary = salary * 1.10, bonus = 1000
          WHERE empno = employee_number;
      ELSEIF (rating = 2)
        THEN UPDATE employee
          SET salary = salary * 1.05, bonus = 500
          WHERE empno = employee_number;
      ELSE UPDATE employee
          SET salary = salary * 1.03, bonus = 0
          WHERE empno = employee_number;
      END IF;
    END
@
```

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

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

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

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

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

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

Customizing precompile and bind options for SQL procedures

The precompile and bind options for SQL procedures can be customized by setting the instance-wide DB2 registry variable, DB2_SQLROUTINE_PREPOPTS with the command:

```
db2set DB2_SQLROUTINE_PREPOPTS=<options>
```

Only the following options are allowed:

```
BLOCKING {UNAMBIG | ALL | NO}
DATETIME {DEF | USA | EUR | ISO | JIS | LOC}
DEGREE {1 | degree-of-parallelism | ANY}
DYNAMICRULES {BIND | RUN | DEFINERUN | DEFINEBIND | INVOKERUN | INVOKEBIND }
EXPLAIN {NO | YES | ALL}
EXPLSNAP {NO | YES | ALL}
FEDERATED {NO | YES}
INSERT {DEF | BUF}
ISOLATION {CS |RR |UR |RS |NC}
QUERYOPT optimization-level
REOPT {ALWAYS |NONE |ONCE}
VALIDATE {RUN | BIND}
```

These options can be changed at the procedure level with the SET_ROUTINE_OPTS stored procedure. The values of the options set for the creation of SQL procedures in the current session can be obtained with the GET_ROUTINE_OPTS function.

Example.

The SQL procedures used in this example will be defined in CLP scripts (given below). These scripts are not in the sqlproc samples directory, but you can easily create these files by cutting-and-pasting the CREATE procedure statements into your own files.

The examples use a table named "expenses", which you can create in the sample database as follows:

```
db2 connect to sample
db2 CREATE TABLE expenses(amount DOUBLE, date DATE)
db2 connect reset
```

To begin, specify the use of ISO format for dates as an instance-wide setting:

```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
db2stop
db2start
```

Stopping and restarting DB2 is necessary for the change to take affect.

Then connect to the database:

```
db2 connect to sample
```

The first procedure is defined in CLP script maxamount.db2 as follows:

```
CREATE PROCEDURE maxamount(OUT maxamnt DOUBLE)
BEGIN
  SELECT max(amount) INTO maxamnt FROM expenses;
END @
```

It will be created with options DATETIME ISO and ISOLATION UR:

```
db2 "CALL SET_ROUTINE_OPTS(GET_ROUTINE_OPTS() || ' ISOLATION UR')"
db2 -td@ -vf maxamount.db2
```

The next procedure is defined in CLP script fullamount.db2 as follows:

```
CREATE PROCEDURE fullamount(OUT fullamnt DOUBLE)
BEGIN
  SELECT sum(amount) INTO fullamnt FROM expenses;
END @
```

It will be created with option ISOLATION CS (note that we are not using the instance-wide DATETIME ISO setting in this case):

```
CALL SET_ROUTINE_OPTS('ISOLATION CS')
db2 -td@ -vf fullamount.db2
```

The last procedure in the example is defined in CLP script perday.db2 as follows:

```
CREATE PROCEDURE perday()
BEGIN
  DECLARE cur1 CURSOR WITH RETURN FOR
    SELECT date, sum(amount)
    FROM expenses
    GROUP BY date;

    OPEN cur1;
END @
```

The last SET_ROUTINE_OPTS call uses the NULL value as the argument. This restores the global setting specified in the DB2_SQLROUTINE_PREPOPTS registry, so the last procedure will be created with option DATETIME ISO:

```
CALL SET_ROUTINE_OPTS(NULL)
db2 -td@ -vf perday.db2
```

Improving the performance of SQL procedures

Overview of how DB2 compiles SQL PL and inline SQL PL

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

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

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

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

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

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

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

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

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

Here is the rewritten SQL function:

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

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

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

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

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

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

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

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

can be rewritten as a single statement:

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

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

Similarly, a sequence of SET statements like:

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

can be written as a single VALUES statement:

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

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

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

Converting the previous two statements to:

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

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

Reduce multiple SQL statements to a single SQL expression

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

Consider the following fragment of SQL PL code:

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

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

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

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

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

can be rewritten as:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Keep the DB2 optimizer informed

When a procedure is created, its individual SQL queries are compiled into sections in a package. The DB2 optimizer chooses an execution plan for a query based, among other things, on table statistics (for example, table sizes or the relative frequency of data values in a column) and indexes available at the time the query is compiled. When tables suffer significant changes, it may be a good idea to let

DB2 collect statistics on these tables again. And when statistics are updated or new indexes are created, it may also be a good idea to rebind the packages associated with SQL procedures that use the tables, to let DB2 create plans that exploit the latest statistics and indexes.

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

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

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

Use arrays

You can use arrays to efficiently pass collections of data between applications and stored procedures and to store and manipulate transient collections of data within SQL procedures without having to use relational tables. Operators on arrays available within SQL procedures allow for the efficient storage and retrieval of data. Applications that create arrays of moderate size will experience significantly better performance than applications that create very large arrays (on the scale of multiple megabytes), as the entire array is stored in main memory. See *Related links* section for additional information.

Chapter 5. Overview of external routines

External routines are characterized primarily by the fact that their routine logic is implemented in programming language code and not in SQL.

Before deciding to implement an external routine, it is important that you understand what external routines are, how they are implemented, and how they can be used. The following concept topics will help you get an understanding of external routines so that you can make informed decisions about when and how to use them in your database environment:

- "External routine features"
- "External routine creation" on page 112
- "External routine library and class management" on page 113
- "Supported APIs and programming languages for external routine development" on page 21
- "32-bit and 64-bit support for external routines" on page 123
- "External routine parameter styles" on page 124
- "Restrictions on external routines" on page 127

Once you have an understanding of external routine concepts you might want to refer to:

- "Creating external routines" on page 143

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 as well as support for additional features not supported by SQL routines. The following features are unique to external routines:

Access to files, data, and applications residing outside of the database
External routines can access and manipulate data or files that reside outside of the database itself. They can also invoke applications that reside

© IBM Corporation 1993, 2008

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 provide support for the passing of additional database and routine property information to and from the routine in a structure named *dbinfo* structure that might be useful within the routine logic.

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. This is done with a structure called a *scratchpad*. This can be useful both for functions that return aggregated values and for 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 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.

External routine creation

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

There are two parts to an external routine:

- The CREATE statement that defines the routine.
- The external library or class that implements the routine-body

Upon the successful execution of a CREATE statement that defines a routine, the routine is created within the database. The statement must at a minimum define the name of the routine, the routine parameter signature that will be used in the routine implementation, and the location of the external library or class built from the routine implementation source code.

External routine implementation must be coded in one of the supported programming languages and then built into a library or class file that must be installed in the file system of the database server.

An external routine cannot be successfully invoked until it has been created in the database and the library or class associated with the routine has been put in the location specified by the EXTERNAL clause.

The development of external routines generally consists of the following tasks:

- Determining what functional type of routine to implement.
- Choosing one of the supported external routine programming languages for the routine implementation.

- Designing the routine.
- Connecting to a database and creating the routine in the database.
  - This is done by executing one of the CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statements or by using a graphical tool that automates this step.
  - This task, also known as defining or registering a routine, can occur at any time before you invoke the routine, except in the following circumstances:
    - For Java routines that reference an external JAR file or files, the external code and JAR files must be coded and compiled before the routine is created in the database using the routine type specific CREATE statement.
    - Routines that execute SQL statements and refer to themselves directly must be created in the database by issuing the CREATE statement before the external code associated with the routine is precompiled and bound. This also applies to situations where there is a cycle of references, for example, Routine A references Routine B, which references Routine A.
- Coding the routine logic such that it corresponds to the routine definition.
- Building the routine and generating a library or class file.
  - For embedded SQL routines this includes: precompiling , compiling, and link the code as well as binding the routine package to the target database.
  - For non-embedded SQL routines this includes: compiling and linking the code.
- Deploying the library or class file to the database server in the location specified in the routine definition.
- Granting the EXECUTE privilege on the routine to the routine invoker or invokers (if they are not the routine definer).
- Invoking, testing, and debugging the routine.

The steps required to create an external routine can all be done using the DB2 Command Line Processor or a DB2 Command Window. Tools can be of assistance in automating some or all of these steps.

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

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 DB2 *function directory* or a sub-directory of this directory on the database server. This is the recommended external routine deployment location. 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 used to implement the routine, however this is generally discouraged. If this is done, to successfully invoke the routine you must take particular note of the fully qualified 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 computer where a DB2 client is installed to a DB2 database server using special system-defined procedures designed specifically for this purpose. See the topics on Java routines for more details.

When executing 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.

By default DB2 will look 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.

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 DB2 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

DB2 external routine library resolution is performed at the DB2 instance level. This means that in DB2 instances containing multiple DB2 databases, external routines can be created in one database that use external routine libraries already being used for a routine 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 DB2, 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 DB2 to load the new classes. If you do not issue the CALL SQLJ.REFRESH_CLASSES() statement after you update Java routine classes, DB2 continues to use the previous versions of the classes. DB2 refreshes the classes when a COMMIT or ROLLBACK occurs.

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 impact routine performance, because the DB2 database manager dynamically caches external routine libraries in an effort to improve performance in accordance with routine usage. 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 re-load the routine's library.

Supported APIs and programming languages for external routine development

You can develop DB2 external routines (procedures and functions) using the following APIs and associated programming languages:

- 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 10. 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 implementation go quickly. • SQL Procedural Language (SQL PL) elements allow for control-flow logic around SQL operations and queries. • Strong data type support.	• 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 10. 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 10. 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 DB2 instances, however these routines will not perform as well as 64-bit COBOL procedures within a 64-bit DB2 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 10. 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	• 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 10. 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 10. 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 DB2 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.

32-bit and 64-bit support for external routines

Support for 32-bit and 64-bit external routines is determined by the specification of one of the following two clauses in the CREATE statement for the routines: FENCED clause or NOT FENCED clause.

The routine-body of an external routine is written in a programming language and compiled into a library or class file that is loaded and run when the routine is invoked. The specification of the FENCED or NOT FENCED clause determines whether the external routine runs in a fenced environment distinct from the database manager or in the same addressing space as the database manager which can yield better performance through the use of shared memory instead of TCPIP for communications. By default routines are always created as fenced regardless of the other clauses selected.

The following table illustrates DB2's support for running fenced and unfenced 32-bit and 64-bit routines on 32-bit and 64-bit database servers that are running the same operating system.

Table 11. Support for 32-bit and 64-bit external routines

Bit-width of routine	32-bit server	64-bit server
32-bit fenced procedure or UDF	Supported	Supported
64-bit fenced procedure or UDF	Not supported (4)	Supported
32-bit unfenced procedure or UDF	Supported	Supported (2)
64-bit unfenced procedure or UDF	Not supported (4)	Supported

The footnotes in the table above correspond to:

- (1) Running a 32-bit routine on a 64-bit server is not as fast as running a 64-bit routine on a 64-bit server.
- (2) 32-bit routines must be created as FENCED and NOT THREADSAFE to work on a 64-bit server.
- (3) It is not possible to invoke 32-bit routines on Linux IA 64-bit database servers.
- (4) 64-bit applications and routines cannot be run in 32-bit addressing spaces.

The important thing to note in the table is that 32-bit unfenced procedures cannot run on a 64-bit DB2 server. If you must deploy 32-bit unfenced routines to 64-bit platforms, remove the NOT FENCED clause from the CREATE statements for these routines before you catalog them.

Performance of routines with 32-bit libraries on 64-bit database servers

It is possible to invoke routines with 32-bit routine libraries on 64-bit DB2 database servers. However, this does not perform as well as invoking a 64-bit routine on a 64-bit server. The reason for the performance degradation is that before each attempt to execute a 32-bit routine on a 64-bit server, an attempt is first made to invoke it as a 64-bit library. If this fails, the library is then invoked as a 32-bit library. A failed attempt to invoke a 32-bit library as a 64-bit library produces an error message (SQLCODE -444) in the db2diag.log.

Java classes are bit-width independent. Only Java virtual machines (JVMs) are classified as 32-bit or 64-bit. DB2 only supports the use of JVMs that are the same bit width as the instance in which they are used. In other words, in a 32-bit DB2 instance only a 32-bit JVM can be used, and in a 64-bit DB2 instance only a 64-bit JVM can be used. This ensures proper functioning of Java routines and the best possible performance.

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 will be passed to the external routine implementation. They also specify what if any additional values will be passed to the external routine implementation. For example, some parameter styles

specify that for each routine parameter value that an additional separate null-indicator value be passed to the routine implementation to provide information about the parameters nullability which cannot otherwise be easily determined with a native programming language data type.

The table below 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 12. Parameter styles

Parameter style	Supported language	Supported routine type	Description
SQL [1]	• C/C++ • OLE • .NET common language runtime languages • COBOL [2]	• UDFs • stored procedures • methods	In addition to the parameters passed during invocation, the following arguments are passed to the routine in the following order: • A null indicator for each parameter or result declared in the CREATE statement. • The SQLSTATE to be returned to DB2. • The qualified name of the routine. • The specific name of the routine. • The SQL diagnostic string to be returned to DB2. Depending on options specified in the CREATE statement and the routine type, the following arguments can be passed to the routine in the following order: • A buffer for the scratchpad. • The call type of the routine. • The dbinfo structure (contains information about the database).
DB2SQL [1]	• C/C++ • OLE • .NET common language runtime languages • COBOL	• stored procedures	In addition to the parameters passed during invocation, the following arguments are passed to the stored procedure in the following order: • A vector containing a null indicator for each parameter on the CALL statement. • The SQLSTATE to be returned to DB2. • The qualified name of the stored procedure. • The specific name of the stored procedure. • The SQL diagnostic string to be returned to DB2. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure.

Table 12. 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 will be passed as single entry arrays to facilitate the returning of values. In addition to the IN, OUT, and INOUT parameters, Java method signatures for stored procedures include a parameter of type ResultSet[] for each result set specified in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement. For PARAMETER STYLE JAVA UDFs and methods, no additional arguments to those specified in the routine invocation are passed. PARAMETER STYLE JAVA routines do not support the DBINFO or PROGRAM TYPE clauses. For UDFs, PARAMETER STYLE JAVA can only be specified when there are no structured data types specified as parameters and no structured type, CLOB, DBCLOB, or BLOB data types specified as return types (SQLSTATE 429B8). Also, PARAMETER STYLE JAVA UDFs do not support table functions, call types, or scratchpads.
DB2GENERAL	• Java	• UDFs • stored procedures • methods	This type of routine will use a parameter passing convention that is defined for use with Java methods. Unless you are developing table UDFs, UDFs with scratchpads, or need access to the dbinfo structure, it is recommended that you use PARAMETER STYLE JAVA. For PARAMETER STYLE DB2GENERAL routines, no additional arguments to those specified in the routine invocation are passed.
GENERAL	• C/C++ • .NET common language runtime languages • COBOL	• stored procedures	A PARAMETER STYLE GENERAL stored procedure receives parameters from the CALL statement in the invoking application or routine. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. GENERAL is the equivalent of SIMPLE stored procedures for DB2 Universal Database for z/OS and OS/390.
GENERAL WITH NULLS	• C/C++ • .NET common language runtime languages • COBOL	• stored procedures	A PARAMETER STYLE GENERAL WITH NULLS stored procedure receives parameters from the CALL statement in the invoking application or routine. Also included is a vector containing a null indicator for each parameter on the CALL statement. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 Universal Database for z/OS and OS/390.

Note:

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

Restrictions on external routines

The following restrictions apply to external routines and should be considered when developing or debugging external routines.

Restrictions that apply to all external routines:

- New threads 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 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: DB2 does not attempt to synchronize any external input or output performed by a routine with DB2's own transactions. So, for example, if a UDF writes to a file during a transaction, and that transaction is later backed out for some reason, no attempt is made to discover or undo the writes to the file.
- Connection-related statements or commands cannot be executed in external routines. This restriction applies to the following statements: including:
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- Operating system function usage within routines is not recommended. The use of these functions is not restricted except 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 DB2's 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 DB2; 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.

- External routines must not contain commands that would terminate the current process. An external routine must always return control to the DB2 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 DB2 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 'DB2' 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 'DB2' 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 DB2 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 DB2 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 DB2 database server. The DB2 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. DB2 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.

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 DB2 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.

Chapter 6. Developing external routines

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"
- "External scalar function and method processing model" on page 133
- "External table functions" on page 133
- "External table function processing model" on page 134
- "Table function execution model for Java" on page 135
- "Scratchpads for external functions and methods" on page 136
- "Scratchpads on 32-bit and 64-bit operating systems" on page 140

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 DB2 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 DB2 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.

© Copyright IBM Corp. 1993, 2008

- 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 DB2 built-in functions.
- Perform logic inside an SQL statement that SQL cannot natively perform.
- Encapsulate a scalar query that is commonly reused as a subquery in SQL statements. For example, given a postal code, search a table for the city where the postal code is found.

Supported languages

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

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

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

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

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

Note: This model describes the ordinary error processing for methods and scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model cannot be made. For example, for a FENCED UDF, if the `db2udf` fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

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 DB2 and the UDF is one-row-at-a-time. There are five types of calls made to a table function: OPEN, FETCH, CLOSE, FIRST, and FINAL. The existence of FIRST and FINAL calls depends on how you define the UDF. The same *call-type* mechanism that can be used for scalar functions is used to distinguish these calls.
- Not every result column defined in the RETURNS clause of the CREATE FUNCTION statement for the table function has to be returned. The DBINFO keyword of CREATE FUNCTION, and corresponding *dbinfo* argument enable the optimization that only those columns needed for a particular table function reference need be returned.
- The individual column values returned conform in format to the values returned by scalar functions.

- The CREATE FUNCTION statement for a table function has a CARDINALITY specification. This specification enables the definer to inform the DB2 optimizer of the approximate size of the result so that the optimizer can make better decisions when the function is referenced.

  Regardless of what has been specified as the CARDINALITY of a table function, exercise caution against writing a function with infinite cardinality, that is, a function that always returns a row on a FETCH call. There are many situations where DB2 expects the end-of-table condition, as a catalyst within its query processing. Using GROUP BY or ORDER BY are examples where this is the case. DB2 cannot form the groups for aggregation until end-of-table is reached, and it cannot sort until it has all the data. So a table function that never returns the end-of-table condition (SQL-state value '02000') can cause an infinite processing loop if you use it with a GROUP BY or ORDER BY clause.

External table function processing model

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

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

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

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

CLOSE call
: This call is made at the conclusion of the scan or statement, provided that the OPEN call succeeded. Any argument values will not be current. The function can return an error.

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

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

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

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

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

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

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

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

Table function execution model for Java

For table functions written in Java and using PARAMETER STYLE DB2GENERAL, it is important to understand what happens at each point in DB2's processing of a given statement. The following table details this information for a typical table function. Covered are both the NO FINAL CALL and the FINAL CALL cases, assuming SCRATCHPAD in both cases.

Point in scan time	**NO FINAL CALL LANGUAGE JAVA SCRATCHPAD**	**FINAL CALL LANGUAGE JAVA SCRATCHPAD**
Before the first OPEN for the table function	No calls.	• Class constructor is called (means new scratchpad). UDF method is called with FIRST call. • Constructor initializes class and scratchpad variables. Method connects to Web server.

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
At each OPEN of the table function	• Class constructor is called (means new scratchpad). UDF method is called with OPEN call. • Constructor initializes class and scratchpad variables. Method connect to Web server, and opens the scan for Web data.	• UDF method is opened with OPEN call. • Method opens the scan for whatever Web data it wants. (Might be able to avoid reopen after a CLOSE reposition, depending on what is saved in the scratchpad.)
At each FETCH for a new row of table function data	• UDF method is called with FETCH call. • Method fetches and returns next row of data, or EOT.	• UDF method is called with FETCH call. • Method fetches and returns new row of data, or EOT.
At each CLOSE of the table function	• UDF method is called with CLOSE call. `close()` method if it exists for class. • Method closes its Web scan and disconnects from the Web server. `close()` does not need to do anything.	• UDF method is called with CLOSE call. • Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist.
After the last CLOSE of the table function	No calls.	• UDF method is called with FINAL call. `close()` method is called if it exists for class. • Method disconnects from the Web server. `close()` method does not need to do anything.

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.
2. For table functions with NO SCRATCHPAD specified, the calls to the UDF method are as indicated in this table, but because the user is not asking for any continuity with a scratchpad, DB2 will cause a new object to be instantiated before each call, by calling the class constructor. It is not clear that table functions with NO SCRATCHPAD (and thus no continuity) can do useful things, but they are supported.

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 DB2. This retrieval is a costly process, so the function would like to do it only one time, and save the list somewhere handy for subsequent calls.

   - On each call.

     Use the list of document IDs saved during the first call to see if the document ID that is passed as the second argument is contained in the list.

   This type of routine is DETERMINISTIC. Its answer only depends on its input argument values. What is shown here is a function whose performance, not correctness, depends on the ability to save information from one call to the next.

Both of these needs are met by the ability to specify a SCRATCHPAD in the CREATE statement:

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

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

The SCRATCHPAD keyword tells DB2 to allocate and maintain a scratchpad for a routine. The default size for a scratchpad is 100 bytes, but you can determine the size (in bytes) for a scratchpad. The *match* example is 10000 bytes long. DB2 initializes the scratchpad to binary zeros before the first invocation. If the scratchpad is being defined for a table function, and if the table function is also defined with NO FINAL CALL (the default), DB2 refreshes the scratchpad before each OPEN call. If you specify the table function option FINAL CALL, DB2 does not examine or change the content of the scratchpad after its initialization. For scalar functions defined with scratchpads, DB2 also does not examine or change the scratchpad's content after its initialization. A pointer to the scratchpad is passed to the routine on each invocation, and DB2 preserves the routine's state information in the scratchpad.

So for the *counter* example, the last value returned could be kept in the scratchpad. And the *match* example could keep the list of documents in the scratchpad if the

scratchpad is big enough, otherwise it could allocate memory for the list and keep the address of the acquired memory in the scratchpad. Scratchpads can be variable length: the length is defined in the CREATE statement for the routine.

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

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

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

Since the scratchpad is of a fixed size, the UDF or method can itself include a memory allocation and thus, can make use of the final call to free the memory. For example, the preceding *match* function cannot predict how many documents will match the given text string. So a better definition for *match* is:

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

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

If you do specify FINAL CALL, 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 DB2 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 take care in the way you create and use scratchpads that contain 64-bit values. It is recommended that you do not declare an explicit length variable for a scratchpad structure that contains one or more 64-bit values, such as 64-bit pointers or `sqlint64` BIGINT variables.

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 */
   }
   ```

Since 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.

XML data type support in external routines

External procedures and functions written in the following programming languages support parameters and variables of data type XML:

- C
- C++
- COBOL
- Java

- .NET CLR languages

External OLE and OLEDB routines do not support parameters 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 CREATE PROCEDURE statement below 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 example below:

```
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.

Writing routines

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

There are some routine features that are specific to a routine type. For example, result sets are specific to stored procedures, and scratchpads are specific to UDFs

and methods. When you come across a step not applicable to the type of routine you are developing, go to the step that follows it.

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.

To create a routine body, you must:

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

   For example, the following is a signature of a UDF body written in C (using sqludf.h) for PARAMETER STYLE SQL:

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

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

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

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.

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.

Prerequisites

- Knowledge of external routine implementation. To learn about external routines in general, see the topic:
  - "External routines" on page 111
  - "External routine creation" on page 112
- The DB2 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.

For a list of restrictions associated with external routines see:

- "Restrictions on external routines" on page 127

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 referenced in the Prerequisites section.
   - Use or import any required header files 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 124
3. Build your code into a library or class file.

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

   You can copy the assembly to another directory on the server if you wish, but to successfully invoke the routine you must note the fully qualified path name of your assembly as you will require it for the next step.
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.

     By default DB2 will look 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 and it will return one or more result sets to the caller.
   - Specify any other clauses required to characterize the routine.

To invoke your external routine, see Routine invocation

.NET common language runtime (CLR) routines

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

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

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

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

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

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

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

Before developing a CLR routine, it is important to both understand the basics of routines and the unique features and characteristics specific to CLR routines. To learn more about routines and CLR routines see:

- "Benefits of using routines" on page 1
- "SQL data type representation in .NET CLR routines" on page 146
- "Parameters in .NET CLR routines" on page 149
- "Returning result sets from .NET CLR procedures" on page 151
- "Restrictions on .NET CLR routines" on page 153
- "Errors related to .NET CLR routines" on page 163

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 DB2 Command Window" on page 155
- "Examples of C# .NET CLR procedures" on page 165
- "Examples of C# .NET CLR functions" on page 199

Support for external routine development in .NET CLR languages

To develop external routines in .NET CLR languages and successfully run them, you will need to use supported operating systems, versions of DB2 database servers and clients, and development software.

Supported operating systems for developing .NET CLR routines with .NET Framework Versions 1.1, 2.0, or 3.0

- Windows 2000
- Windows XP (32-bit edition)
- Windows Server 2003 (32-bit edition)

Supported DB2 database servers and clients for .NET CLR routine development

The following minimum DB2 database servers and clients must be installed:

- DB2 server: Minimum supported version is DB2 Version 8.2.
- DB2 client: Minimum supported version is DB2 Version 7.2.

Required development software for .NET CLR routines

One of the following two software products must be installed on the same computer as the DB2 database server:

- Microsoft .NET Framework, Version 1.1
- Microsoft .NET Framework, Version 2.0

The Microsoft .NET Framework is independently available or as part of one of the following development kits:

- Microsoft .NET Framework Version 1.1 Software Development Kit
- Microsoft .NET Framework Version 2.0 Software Development Kit
- Microsoft .NET Framework Version 3.0 Software Development Kit

.NET CLR external routines can be implemented in any language that can be compiled into an IL assembly by the Microsoft .NET Framework. These languages include, but are not limited to: Managed C++, C#, Visual Basic, and J#.

Tools for developing .NET CLR routines

Tools can make the task of developing .NET CLR routines that interact with DB2 database faster and easier.

.NET CLR routines can be developed in Microsoft Visual Studio .NET using graphical tools available in:

- IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2

The following command line interfaces, provided with DB2, are also available for developing .NET CLR routines on DB2 database servers:

- DB2 Command Line Processor (DB2 CLP)
- DB2 Command 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 routines

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

- .NET CLR routines

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, you might want to: ″Create .NET CLR routines″.

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, DB2 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 DB2 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, DB2 data types, Informix data types, Microsoft .NET Framework types, and DB2Types classes and structures.

Category	DB2Types Classes and Structures	DB2Type Data Type	DB2 Data Type	Informix Data Type	.NET Data Type
Numeric	DB2Int16	SmallInt	SMALLINT	BOOLEAN, SMALLINT	Int16
	DB2Int32	Integer	INT	INTEGER, INT, SERIAL	Int32
	DB2Int64	BigInt	BIGINT	INT8, SERIAL8	Int64
	DB2Real, DB2Real370	Real	REAL	REAL, SMALLFLOAT	Single
	DB2Double	Double	DOUBLE PRECISION	DECIMAL (≤31), DOUBLE PRECISION	Double
	DB2Double	Float	FLOAT	DECIMAL (32), FLOAT	Double
	DB2Decimal	Decimal	DECIMAL	MONEY	Decimal
	DB2DecimalFloat	DecimalFloat	DECFLOAT(16 | 34)[4]		Decimal
	DB2Decimal	Numeric	DECIMAL	DECIMAL (≤31), NUMERIC	Decimal
Date/Time	DB2Date	Date	DATE	DATETIME (date precision)	Datetime
	DB2Time	Time	TIME	DATETIME (time precision)	TimeSpan
	DB2TimeStamp	Timestamp	TIMESTAMP	DATETIME (time and date precision)	DateTime
XML	DB2Xml	Xml[2]	XML		Byte[]
Character data	DB2String	Char	CHAR	CHAR	String
	DB2String	VarChar	VARCHAR	VARCHAR	String
	DB2String	LongVarChar[1]	LONG VARCHAR	LVARCHAR	String
Binary data	DB2Binary	Binary	CHAR FOR BIT DATA		Byte[]
	DB2Binary	Binary[3]	BINARY		Byte[]
	DB2Binary	VarBinary[3]	VARBINARY		Byte[]
	DB2Binary	LongVarBinary[1]	LONG VARCHAR FOR BIT DATA		Byte[]
Graphic data	DB2String	Graphic	GRAPHIC		String
	DB2String	VarGraphic	VARGRAPHIC		String
	DB2String	LongVarGraphic[1]	LONG VARGRAPHIC		String

1. These data types are not supported as parameters in DB2 .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 UDB for z/OS.
4. This data type is only supported for DB2 for z/OS Version 9 and later releases and for DB2 for Linux, UNIX, and Windows Version 9.5 and later releases.

Category	DB2Types Classes and Structures	DB2Type Data Type	DB2 Data Type	Informix Data Type	.NET Data Type
LOB data	DB2Clob	Clob	CLOB	CLOB, TEXT	String
	DB2Blob	Blob	BLOB	BLOB, BYTE	Byte[]
	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 DB2 parameter styles are supported for CLR routines:

- `SQL` (Supported for procedures and functions)
- `GENERAL` (Supported for procedures only)
- `GENERAL WITH NULLS` (Supported for procedures only)
- `DB2SQL` (Supported for procedures and functions)

For more information about these parameter styles see the topic:

- Parameter styles for external routines

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 below.

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

- Parameter styles for external routines

Passing CLR routine parameters by value or by reference

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

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

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

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

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

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

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

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

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

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

No parameter marker is required for procedure result sets

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

For more on result sets in CLR routines, see:

•

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).

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 follow the steps in the procedure below for returning results from a CLR procedure.

- "Creating .NET CLR routines from DB2 Command Window" on page 155

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 application developer, you might want to protect the assemblies associated with your DB2 external routines from unwelcome tampering to restrict the actions of routines at run time. DB2 .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, DB2 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 DB2 detects at run time that a CLR routine is attempting an action outside of the scope of its execution control mode, DB2 will return 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. These restrictions are listed here.

The CREATE METHOD statement with LANGUAGE CLR clause is not supported

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

CLR procedures cannot be implemented as NOT FENCED procedures

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

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 DB2 is represented with a precision of 31 digits and a scale of 28 digits. The .NET CLR System.Decimal data type is limited to a precision of 29 digits and a scale of 28 digits. Therefore, DB2 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. DB2 will raise 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, DB2 will raise an error (SQLSTATE 42613, SQLCODE -628).

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

Data types not supported in CLR routines

The following DB2 SQL data types are not supported in CLR routines:

- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- LONG GRAPHIC
- 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 DB2 database server
- Developing the routine implementation that corresponds to the routine definition

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

- Using the graphical tools provided with the DB2 Database Development Add-In for Visual Studio .NET 1.2
- Using the DB2 Command Window

In general it is easiest to create .NET CLR routines using the DB2 Database Development Add-In for Visual Studio .NET 1.2. If this is not available for use, the DB2 Command Window provides similar support through a command line interface.

Prerequisites

- Review the ".NET common language runtime (CLR) routines" on page 144.
- Ensure that you have access to a DB2 Version 9 server, including instances and databases.
- Ensure that the operating system is at a version level that is supported by DB2 database products.
- Ensure that the Microsoft .NET development software is at a version level that is supported for .NET CLR routine development.
- Authority to execute the CREATE PROCEDURE or CREATE FUNCTION statement.

For a list of restrictions associated with CLR routines see:

- "Restrictions on .NET CLR routines" on page 153

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

- Visual Studio .NET when the IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2 is also installed. When the Add-In is installed, graphical tools integrated into Visual Studio .NET are available for creating .NET CLR routines that work in DB2 database servers.
- DB2 Command Window

To create .NET CLR routines from DB2 Command Window, see:

- "Creating .NET CLR routines from DB2 Command Window"

Creating .NET CLR routines from DB2 Command Window

Procedures and functions that reference an intermediate language assembly are created in the same way as any external routine is created. You would choose to implement an external routine in a .NET language if:

- You want to encapsulate complex logic into a routine that accesses the database or that performs an action outside of the database.
- You require the encapsulated logic to be invoked from any of: multiple applications, the CLP, another routine (procedure, function (UDF), or method), or a trigger.
- You are most comfortable coding this logic in a .NET language.

Prerequisites

- Knowledge of CLR routine implementation. To learn about CLR routines in general and about CLR features, see:
  - ".NET common language runtime (CLR) routines" on page 144
- The database server must be running a Windows operating system that supports the Microsoft .NET Framework.
- The .NET Framework, version 1.1 or 2.0, must be installed on the server. The .NET Framework is independently available or as part of the Microsoft .NET Framework 1.1 Software Development Kit or .NET Framework 2.0 Software Development Kit.
-

The following versions of DB2 must be installed:
  - Server: DB2 8.2 or a later release.
  - Client: Any client that can attach to a DB2 8.2 instance will be able to invoke a CLR routine. It is recommended that you install DB2 Version 7.2 or a later release on the client.
- Authority to execute the CREATE statement for the external routine. For the privileges required to execute the CREATE PROCEDURE statement or CREATE FUNCTION statement, see the details of the appropriate statement.

For a list of restrictions associated with CLR routines see:
- "Restrictions on .NET CLR routines" on page 153

Procedure

1. Code the routine logic in any CLR supported language.
   - For general information about .NET CLR routines and .NET CLR routine features see the topics referenced in the Prerequisites section
   - Use or import the `IBM.Data.DB2` assembly if your routine will execute SQL.
   - Declare host variables and parameters correctly using data types that map to DB2 SQL data types. For a data type mapping between DB2 and .NET data types:
     - "SQL data type representation in .NET CLR routines" on page 146
   - Parameters and parameter null indicators must be declared using one of DB2's supported parameter styles and according to the parameter requirements for .NET CLR routines. As well, scratchpads for UDFs, and the DBINFO class are passed into CLR routines as parameters. For more on parameters and prototype declarations see:
     - "Parameters in .NET CLR routines" on page 149
   - 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 151
   - Set a routine return value if required. CLR scalar functions require that a return value is set before returning. CLR table functions require that a return code is specified as an output parameter for each invocation of the table function. CLR procedures do not return with a return value.
2. Build your code into an intermediate language (IL) assembly to be executed by the CLR. For information on how to build CLR .NET routines that access DB2, see the following topic:
   - "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*
3. Copy the assembly into the DB2 *function directory* on the database server. It is recommended that you store assemblies or libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the assembly to another directory on the server if you 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 DB2 will look for the assembly by name in the function directory unless a fully qualified or relative path name for the library is specified in the EXTERNAL clause.

  When the CREATE statement is executed, if the assembly specified in the EXTERNAL clause is not found by DB2 you will receive an error (SQLCODE -20282) with reason code 1.
- Specify 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 has been written, it must be built before the routine assembly can be deployed and the routine invoked. The steps required to build .NET CLR routines are similar to those required to build any external routine however there are some differences.

There are three ways to build .NET CLR routines:

- Using the graphical tools provided with the DB2 Database Development Add-In for Visual Studio .NET 1.2
- Using DB2 sample batch files
- Entering commands from a DB2 Command Window

The DB2 sample build scripts and batch files for routines are designed for building DB2 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 DB2 sample build scripts and batch files for C# and Visual Basic. In general it is easiest to build .NET CLR routines using the graphical tools or the build scripts which can easily be modified if required, however it is often helpful to know how to build routines from DB2 Command Window as well.

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

Building .NET common language runtime (CLR) routine source code is a sub-task of creating .NET CLR routines. This task can be done quickly and easily using DB2 sample batch files. 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 simplify the task of building .NET CLR routine code by doing so in Visual Studio .NET or you do the steps in the DB2 sample build scripts manually. Refer to:

- Building .NET common language runtime (CLR) routines in Visual Studio .NET

- Building .NET common language runtime (CLR) routines using DB2 Command Window

The programming language specific sample build scripts for building C# and Visual Basic .NET CLR routines are named `bldrtn`. They are located in DB2 directories along with sample programs that can be built with them as follows:

- 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 to the DB2 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. Since 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 DB2 instance in which the routines are to be created.

Procedure

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

1. Open a 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 DB2. 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 DB2 Command Window

Building .NET CLR routine source code is a sub-task of creating .NET CLR routines. This task can be done manually from DB2 Command Window. The same procedure can be followed regardless of whether there are SQL statements within the routine code or not. The task steps include compilation of source code written in a .NET CLR supported programming language into an assembly with a .DLL file suffix.

As alternatives, you can simplify the task of building .NET CLR routine code by doing so in Visual Studio .NET or by using DB2 sample build scripts. Refer to:

- Building .NET common language runtime (CLR) routines in Visual Studio .NET
- Building .NET common language runtime (CLR) routines using sample build scripts

Prerequisites

- Required operating system and .NET CLR routine development software prerequisites have been satisfied. See: ″Support for .NET CLR routine development″.
- Source code written in a supported .NET CLR programming language containing one or more .NET CLR routine implementations.
- The name of the database within the current DB2 instance in which the routines are to be created.
- The operating specific compile and link options required for building .NET CLR routines.

To build a source code file that contains one or more .NET CLR routine code implementations, follow the steps below. An example follows that demonstrates each of the steps:

1. Open a 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 will be created.
4. Compile the source code file.
5. Link the source code file to generate a shared library. This requires the use of some DB2 specific compile and link options.
6. Copy the assembly file with the .DLL file suffix to the DB2 function directory on the database server.
7. 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 DB2. You can do this by issuing the db2stop command followed by the db2start command.

Once you have successfully built and deployed the routine library, you should complete the steps associated with the task of creating .NET CLR routines. Creating .NET CLR routines includes a step for executing the CREATE statement for each routine that was implemented in the source code file. This step must also be completed before you will be able to invoke the routines.

Example

The following example demonstrates the re-building of a .NET CLR source code file. Steps are shown for both a Visual Basic code file named `myVBfile.vb` containing routine implementations as well as for a C# code file named `myCSfile.cs`. The routines are being built on a Windows 2000 operating system using Microsoft .NET Framework 1.1 to generate a 64-bit assembly.

1. Open a 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 will be created.

   ```
   db2 connect to <database-name>
   ```

4. Compile the source code file using the recommended compile and link options (where $DB2PATH is the install path of the DB2 instance. Replace this value before running the command):

   ```
   C# example
   ==================
   csc /out:myCSfile.dll /target:library
       /reference:$DB2PATH%\bin\netf11\IBM.Data.DB2.dll myCSfile.cs

   Visual Basic example
   ====================
   vbc /target:library /libpath:$DB2PATH\bin\netf11
       /reference:$DB2PATH\bin\netf11\IBM.Data.DB2.dll
       /reference:System.dll
       /reference:System.Data.dll myVBfile.vb
   ```

   The compiler will generate output if there are any errors. This step generates an export file named myfile.exp.

5. Copy the shared library to the DB2 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 DB2 looks for routine libraries. Refer to the topic on creating .NET CLR routines for more on deploying routine libraries.

6. Stop and restart the database as this is a re-building of a previously built routine source code file.

   ```
   db2stop
   db2start
   ```

Building .NET CLR routines is generally most easily done using the operating specific sample build scripts which also can be used as a reference for how to build routines from the command line.

CLR .NET routine compile and link options

The following are the compile and link options recommended by DB2 for building Common Language Runtime (CLR) .NET routines on Windows with either the Microsoft Visual Basic .NET compiler or the Microsoft C# compiler, as demonstrated in the `samples\.NET\cs\bldrtn.bat` and `samples\.NET\vb\bldrtn.bat` batch files.

Compile and link options for bldrtn
Compile and link options using the Microsoft C# compiler: **csc** The Microsoft C# compiler. **/out:%1.dll /target:library** Output the dynamic link library as a stored procedure assembly dll. **/debug** Use the debugger. **/lib: "%DB2PATH%"\bin\netf20** Use the library path for .NET Framework Version 2.0. There are three supported versions of the .NET framework for applications: version 1.1, version 2.0, and version 3.0. There is a dynamic link library for each in separate sub-directories. For .NET Framework Version 1.1, use the `"%DB2PATH%"\bin\netf11` sub-directory. For .NET Framework Version 2.0 and 3.0, use the `"%DB2PATH%"\bin\netf20` sub-directory. **/reference:IBM.Data.DB2.dll** Use the DB2 dynamic link library for the IBM Data Server Provider for .NET Refer to your compiler documentation for additional compiler options.
Compile and link options using the Microsoft Visual Basic .NET compiler: **vbc** The Microsoft Visual Basic .NET compiler. **/out:%1.dll /target:library** Output the dynamic link library as a stored procedure assembly dll. **/debug** Use the debugger. **/libpath:"%DB2PATH%"\bin\netf20** Use the library path for .NET Framework Version 2.0. There are three supported versions of the .NET framework for applications: version 1.1, version 2.0, and version 3.0. There is a dynamic link library for each in separate sub-directories. For .NET Framework Version 1.1, use the `"%DB2PATH%"\bin\netf11` sub-directory. For .NET Framework Version 2.0 and 3.0, use the `"%DB2PATH%"\bin\netf20` sub-directory. **/reference:IBM.Data.DB2.dll** Use the DB2 dynamic link library for the IBM Data Server Provider for .NET. **/reference:System.dll** Reference the Microsoft Windows System dynamic link library. **/reference:System.Data.dll** Reference the Microsoft Windows System Data dynamic link library. Refer to your compiler documentation for additional compiler options.

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.

Consider the following when debugging .NET CLR routines:

- Verify that a supported operating system for .NET CLR routine development is being used.
- Verify that both a supported DB2 database server and DB2 client for .NET CLR routine development are being used.
- 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 a user with SYSADM authorization, DBADM authorization, or by the definer of 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 DB2 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 DB2 database managers or DB2 databases.
  - 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.

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

- "Errors related to .NET CLR routines" on page 163

Errors related to .NET CLR routines

Although external routines share a generally common implementation, there are some DB2 errors that might arise that are specific to CLR routines. This reference lists the most commonly encountered .NET CLR related errors listed by their SQLCODE or behavior along with some debugging suggestions. DB2 errors related to routines can be classified as follows:

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

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

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

CLR routine creation time errors

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

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

- `<a>` is the CLR assembly file in which the class is located.
- `<b>` is the class in which the method to invoke resides.
- `<c>` is the method to invoke.

No leading or trailing blank characters are permitted between the single 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 the short form of the file name is used, if the platform is UNIX or if the routine is a LANGUAGE CLR routine, then the file must reside in the function directory. If the platform is Windows and the routine is not a LANGUAGE CLR routine then the file must reside in the system PATH. File extensions (examples: `.a` (on UNIX), `.dll` (on Windows)) should always be included in the file name.

CLR routine runtime errors

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

- Check that the EXTERNAL clause specifies the correct routine assembly name and that the assembly is located in the specified location. If the

EXTERNAL clause does not specify a fully qualified path name to the desired assembly, DB2 presumes that the path name provided is a relative path name to the assembly, relative to the DB2 function directory.

SQLCODE -20282, SQLSTATE 42724, reason code 2

An assembly was found in the location specified by the EXTERNAL clause in the CREATE statement for the routine, but no class was found within the assembly to match the class specified in the EXTERNAL clause.

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

SQLCODE -20282, SQLSTATE 42724, reason code 3

An assembly was found in the location specified by the EXTERNAL clause in the CREATE statement for the routine, that had a correctly matching class definition, but the routine method signature does not match the routine signature specified in the CREATE statement for the routine.

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

SQLCODE -4301, SQLSTATE 58004, reason code 5 or 6

An error occurred while attempting to start or communicate with a .NET interpreter. DB2 was unable to load a dependent .NET library [reason code 5] or a call to the .NET interpreter failed [reason code 6].

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

SQLCODE -4302, SQLSTATE 38501

An unhandled exception occurred while executing, preparing to execute, or subsequent to executing the routine. This could be the result of a routine logic programming error that was unhandled or could be the result of an internal processing error. For errors of this type, the .NET stack traceback that indicates where the unhandled exception occurred will be written to the 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 will be made in the db2diag.log specifically indicating that the exception occurred due to an execution control violation. The exception stack traceback that indicates where the violation occurred will also be included.

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 152

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. 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.

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 199

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

- ".NET common language runtime (CLR) routines" on page 144
- "Creating .NET CLR routines from DB2 Command Window" on page 155
- "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*

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

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

- The C# external code file
- Example 1: C# parameter style GENERAL procedure

- Example 2: C# parameter style GENERAL WITH NULLS procedure
- Example 3: C# parameter style SQL procedure
- Example 4: C# procedure returning a result set
- Example 5: C# procedure accessing the dbinfo structure
- Example 6: C# procedure in PROGRAM TYPE MAIN style

The C# external code file

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

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

Table 13. C# external code file format

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

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

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

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

Example 1: C# parameter style GENERAL procedure

This example shows the following:

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

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

Table 14. Code to create a C# parameter style GENERAL procedure

```
CREATE PROCEDURE setEmpBonusGEN(IN empID CHAR(6), INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
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.

Table 15. Code to create a C# parameter style GENERAL WITH NULLS procedure

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

Table 15. Code to create a C# parameter style GENERAL WITH NULLS procedure (continued)

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

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

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

      reader.Close();
   }
}
```

Example 3: C# parameter style SQL procedure

This example shows the following:

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

This procedure takes an employee ID and a current bonus amount as input. It retrieves the employee's name and salary. If the current bonus amount is zero, a new bonus based on salary is calculated and returned

along with the employee's full name. If the employee is not found, an empty string is returned.

Table 16. Code to create a C# procedure in parameter style SQL with parameters

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

Table 16. Code to create a C# procedure in parameter style SQL with parameters (continued)

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

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

      DB2DataReader reader = myCommand.ExecuteReader();

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

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

      reader.Close();
   }
}
```

Example 4: C# parameter style GENERAL procedure returning a result set

This example shows the following:

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

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

Table 17. Code to create a C# procedure that returns a result set

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

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

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

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

Example 5: C# parameter style SQL procedure accessing the dbinfo structure

This example shows the following:

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

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

Table 18. Code to create a C# procedure that accesses the dbinfo structure

```
CREATE PROCEDURE ReturnDbName(OUT dbName VARCHAR(20))
SPECIFIC ReturnDbName
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE SQL
FENCED
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"
    //
    //  DB2 returns the above values to the client in the
    //  SQLCA structure.  The values are used to generate a
    //  standard DB2 sqlStateate error.
 }
```

Example 6: C# procedure with PROGRAM TYPE MAIN style

This example shows the following:

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

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

Table 19. Code to create a C# procedure in program type MAIN style

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

Table 19. Code to create a C# procedure in program type MAIN style (continued)

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

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

     DB2DataReader reader = myCommand.ExecuteReader();

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

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

     reader.Close();
   }
}
```

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. For examples of CLR procedures in Visual Basic:

- "Examples of Visual Basic .NET CLR procedures" on page 181

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

- ".NET common language runtime (CLR) routines" on page 144
- "Creating .NET CLR routines from DB2 Command Window" on page 155
- "External scalar functions" on page 131
- "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*

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

Use the following examples as references when making your own Visual Basic CLR UDFs:

- The Visual Basic external code file
- Example 1: Visual Basic parameter style SQL table function
- Example 2: Visual Basic parameter style SQL scalar function

The Visual Basic external code file

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

Table 20. Visual Basic external code file format

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

Namespace bizLogic

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

End Namespace
```

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

Example 1: Visual Basic parameter style SQL table function

This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL table function

- Visual Basic code for a parameter style SQL table function

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

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

Table 21. Code to create a Visual Basic parameter style SQL table function

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

Table 21. Code to create a Visual Basic parameter style SQL table function (continued)

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

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

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

    name = newName
    position = newPosition
    salary = newSalary
  End Sub

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

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

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

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

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

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

End Class
```

Table 21. Code to create a Visual Basic parameter style SQL table function (continued)

```
Class empOps

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

    Dim intRow As Int16

    intRow = 0

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

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

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

       Case 1    ' Case SQLUDF_TF_CLOSE:

       Case 2    ' Case SQLUDF_TF_FINAL:
    End Select

  End Sub

End Class
```

Example 2: Visual Basic parameter style SQL scalar function

This example shows the following:

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

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

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

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

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

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

The output of such a query would be:

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

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

Table 22. Code to create a Visual Basic parameter style SQL scalar function

```
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';
```

Table 22. Code to create a Visual Basic parameter style SQL scalar function (continued)

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

     Dim counter As Int32
     counter = 1

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

End Class
```

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. For examples of CLR UDFs in `Visual Basic`:

- "Examples of Visual Basic .NET CLR functions" on page 176

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

- ".NET common language runtime (CLR) routines" on page 144
- "Creating .NET CLR routines from DB2 Command Window" on page 155
- "Benefits of using routines" on page 1
- "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*

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

Use the following examples as references when making your own Visual Basic CLR procedures:

- The Visual Basic external code file
- Example 1: Visual Basic parameter style GENERAL procedure
- Example 2: Visual Basic parameter style GENERAL WITH NULLS procedure
- Example 3: Visual Basic parameter style SQL procedure
- Example 4: Visual Basic procedure returning a result set
- Example 5: Visual Basic procedure accessing the dbinfo structure
- Example 6: Visual Basic procedure in PROGRAM TYPE MAIN style

The Visual Basic external code file

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

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

Table 23. Visual Basic external code file format

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

Namespace bizLogic

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

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

It is important to note the name of the file, the namespace, and the name of the class, that contains a given procedure implementation. These names are important, because the EXTERNAL clause of the CREATE

PROCEDURE statement for each procedure must specify this information so that DB2 can locate the assembly and class of the CLR procedure.

Example 1: Visual Basic parameter style GENERAL procedure

This example shows the following:

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

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

Table 24. Code to create a Visual Basic parameter style GENERAL procedure

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

Table 24. Code to create a Visual Basic parameter style GENERAL procedure (continued)

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

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

   salary = 0

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

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

      salary = myReader.GetDecimal(3)

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

   myReader.Close()

End Sub
```

Example 2: Visual Basic parameter style GENERAL WITH NULLS procedure

This example shows the following:

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

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

Table 25. Code to create a Visual Basic parameter style GENERAL WITH NULLS procedure

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

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

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

     salary = 0

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

        myReader = myCommand.ExecuteReader()

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

           salary = myReader.GetDecimal(3)

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

        myReader.Close()

     End If

  End Sub
```

Example 3: Visual Basic parameter style SQL procedure

This example shows the following:

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

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

Table 26. Code to create a Visual Basic procedure in parameter style SQL with parameters

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

Table 26. Code to create a Visual Basic procedure in parameter style SQL with parameters (continued)

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

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

   salary = 0

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

      myReader = myCommand.ExecuteReader()

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

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

      myReader.Close()
   End If

End Sub
```

Example 4: Visual Basic parameter style GENERAL procedure returning a result set

This example shows the following:

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

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

Table 27. Code to create a Visual Basic procedure that returns a result set

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

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

     Dim myCommand As DB2Command
     Dim myReader As DB2DataReader

     myCommand = DB2Context.GetCommand()

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

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

   End Sub
```

Example 5: Visual Basic parameter style SQL procedure accessing the dbinfo structure

This example shows the following:

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

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

Table 28. Code to create a Visual Basic procedure that accesses the dbinfo structure

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

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

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

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

Example 6: Visual Basic procedure with PROGRAM TYPE MAIN style

This example shows the following:

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

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

Table 29. Code to create a Visual Basic procedure in program type MAIN style

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

Table 29. Code to create a Visual Basic procedure in program type MAIN style (continued)

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

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

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

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

      myReader = myCommand.ExecuteReader()


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

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

      myReader.Close()
   End If

End Sub
```

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 start creating and using CLR procedures with XML features.

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

Prerequisites

Before working 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 DB2 Command Window
- Benefits of using routines
- "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*

The examples below 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# CLR procedures:

- The C# external code file
- Example 1: C# parameter style GENERAL procedure with XML features

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:

Table 30. C# external code file 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 contains 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 provided in the EXTERNAL clause of the CREATE PROCEDURE statement.

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

Example 1: C# parameter style GENERAL procedure with XML features

This example shows the following:

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

This procedure takes two parameters, an integer `inNum` and `inXML`. These values are inserted into the table `xmlDataTable`. Then an XML value is retrieved using XQuery. Another XML value is retrieved using SQL. The retrieved XML values are assigned to two output parameters, `outXML1` and `outXML2`. No result sets are returned.

Table 31. Code to create a C# parameter style GENERAL procedure

```
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
//*************************************************************************
```

Table 31. Code to create a C# parameter style GENERAL procedure (continued)

```
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 example below 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 examples below 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
- Example 1: C parameter style SQL procedure with XML features

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.SQC` and has the following format:

Table 32. C external code file 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 DB2 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.

Table 33. Code to create a C parameter style SQL procedure

```
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
//*************************************************************************
```

Table 33. Code to create a C parameter style SQL procedure (continued)

```
#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. For examples of CLR procedures in C#:

- "Examples of C# .NET CLR procedures" on page 165

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

- ".NET common language runtime (CLR) routines" on page 144
- "Creating .NET CLR routines from DB2 Command Window" on page 155
- "External scalar functions" on page 131
- "Building common language runtime (CLR) .NET routines" in *Developing ADO.NET and OLE DB Applications*

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

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

- The C# external code file
- Example 1: C# parameter style SQL table function
- Example 2: C# parameter style SQL scalar function

The C# external code file

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

Table 34. C# external code file format

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

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

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

Example 1: C# parameter style SQL table function

This example shows the following:

- CREATE FUNCTION statement for a parameter style SQL table function

- C# code for a parameter style SQL table function

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

Table 35. Code to create a C# parameter style SQL table function

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

Table 35. Code to create a C# parameter style SQL table function (continued)

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

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

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

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

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

Table 35. Code to create a C# parameter style SQL table function (continued)

```
class empOps
{

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

      Int16 intRow = 0;

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

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

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

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

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

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

Example 2: C# parameter style SQL scalar function

This example shows the following:

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

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

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

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

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

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

The output of such a query would be:

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

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

Table 36. Code to create a C# parameter style SQL scalar function

```
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' ;
```

Table 36. Code to create a C# parameter style SQL scalar function (continued)

```
class empOps
{
   public static void CountUp(     Int32 input,
                                out Int32 outCounter,
                                    Int16 inputNullInd,
                                out Int16 outCounterNullInd,
                                ref String sqlState,
                                    String funcName,
                                    String specName,
                                ref String sqlMessageText,
                                    Byte[] scratchPad,
                                    Int32 callType)
   {
      Int32 counter = 1;

      switch(callType)
      {
         case -1: // case SQLUDF_FIRST_CALL
           scratchPad[0] = (Byte)counter;
           outCounter = counter;
           outCounterNullInd = 0;
           break;
         case 0:  // case SQLUDF_NORMAL_CALL:
           counter = (Int32)scratchPad[0];
           counter = counter + 1;
           outCounter = counter;
           outCounterNullInd = 0;
           scratchPad[0] =
           (Byte)counter;
           break;
         case 1:  // case SQLUDF_FINAL_CALL:
           counter =
           (Int32)scratchPad[0];
           outCounter = counter;
           outCounterNullInd = 0;
           break;
         default: // Should never enter here
                  // * Required so that at compile time
                  //   out parameter outCounter is always set *
           outCounter = (Int32)(0);
           outCounterNullInd = -1;
           sqlState="ABCDE";
           sqlMessageText = "Should not get here: Default
           case!";
           break;
      }
   }
}
```

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 DB2 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 DB2 will use 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 DB2 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.

The supported compilers and development software for DB2 database application development in C can all be used for external routine development in C++.

Tools for developing C and C++ routines

The tools supported for C and C++ routines are the same as those supported for embedded SQL C and C++ applications.

There are no DB2 development environments or graphical user interface tools for developing, debugging, or deploying embedded SQL applications or routines.

The following command line interfaces are commonly used 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 required to create routines in a database. The PREPARE command and the BIND command required to build C and C++ routines that contain 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.

Prerequisites

- 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 21

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

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

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

- "Include file required for C and C++ routine development (sqludf.h)" on page 207
- "Parameters in C and C++ routines" on page 208
- "Parameter style SQL C and C++ procedures" on page 210
- "Parameter style SQL C and C++ functions" on page 213
- "SQL data type handling in C and C++ routines" on page 223
- "Graphic host variables in C and C++ routines" on page 242

- "Returning result sets from C and C++ procedures" on page 244
- "C++ type decoration" on page 242
- "Restrictions on external routines" on page 127

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

- "Creating C and C++ routines" on page 245

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 DB2 installation include directory that contains this 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:
  - 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. An example is shown below:

```
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 209
- "Parameter null indicators in C and C++ routines" on page 209
- "Parameter style SQL C and C++ procedures" on page 210
- "Parameter style SQL C and C++ functions" on page 213
- "Passing parameters by value or by reference in C and C++ routines" on page 215
- "Parameters are not required for C and C++ procedure result sets" on page 215
- "Dbinfo structure as C or C++ routine parameter" on page 215
- "Scratchpad as C or C++ function parameter" on page 218
- "Program type MAIN support for C and C++ procedures" on page 219

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

The following parameter styles are supported for C and C++ routines:

- SQL (Supported for procedures and functions; recommended)
- GENERAL (Supported for procedures)
- GENERAL WITH NULLS (Supported for 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 124
- “Parameter style SQL C and C++ procedures” on page 210
- “Parameter style SQL C and C++ functions” on page 213

Parameter style SQL C and C++ procedures

C and C++ procedures should be created using the PARAMETER STYLE SQL clause in the CREATE PROCEDURE statement. The parameter passing conventions of this parameter style should be implemented in the corresponding procedure code implementation.

The C and C++ PARAMETER STYLE SQL signature implementation required for procedures follows this format:

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

SQL_API_RC SQL_API_FN

SQL_API_RC and SQL_API_FN are macros that specify the return type and calling convention for a C 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 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 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 should 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 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 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 defined as SQLUDF_NULLIND*.

sqlstate Input-output parameter value 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 DB2 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 formulating diagnostic information including error messages, or when writing to a log file.

specific-name
Input parameter value that contains the unique routine specific name. This value is generated by DB2 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 optionally used by the routine to return message text to the invoking 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.

Note: To simplify the writing of C and C++ procedure signatures the macro definition SQLUDF_TRAIL_ARGS defined in sqludf.h can be used in the procedure signature in place of using individual arguments to implement the non-SQL data type arguments.

The following is an example of a C or C++ procedure implementation that 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 DB2 (output)
             - qualified name of the routine (input)
             - specific name of the routine (input)
             - SQL diagnostic string to return an optional
               error message text to DB2 (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 DB2 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 preceding statement assumes that the C or C++ procedure implementation is in a library file 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. This 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.

Note: DB2 controls the allocation of memory for all parameters and maintains C or C++ references to all parameters passed into or out of a routine. There is no need to allocate or free memory 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 244

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 LANGUAGE C routines through the use of the `sqludf_dbinfo` structure. This C structure is defined in the DB2 include file `sqludf.h` 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   */
   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 this structure's 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.

Here is an example of a C procedure with PARAMETER STYLE GENERAL that demonstrates the use of the `dbinfo` structure. Here is the CREATE PROCEDURE statement for the procedure. Note that as specified by the EXTERNAL NAME clause, the procedure implementation is located in a library file named `spserver` that contains a C function named `DbinfoExample`:

```
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 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. This C structure is defined in the DB2 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 above, 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

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

Table 37. 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*<=254 SQLUDF_CHAR	Fixed-length, null-terminated character string

Table 37. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
CHAR(*n*) FOR BIT DATA	char[*n*] where n is large enough to hold the data 1<=*n*<=254 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

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

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

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

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

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

Table 37. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
TIME	char[9] SQLUDF_TIME	Null-terminated character string of the following format: `hh.mm.ss`
TIMESTAMP	char[27] SQLUDF_STAMP	Null-terminated character string of the following format: `yyyy-mm-dd-hh.mm.ss.nnnnnn`
LOB LOCATOR	sqluint32 SQLUDF_LOCATOR	32-bit signed integer
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 DB2 on argument values. However, if incorrect data types are specified in the routine code, then unpredictable behavior, such as loss of data or abends, will occur.

For the result of a scalar function or method, it is the data type specified in the CAST FROM clause of the CREATE FUNCTION statement that defines the format. If no CAST FROM clause is present, then the data type specified in the RETURNS clause defines the format.

In the following example, the presence of the CAST FROM clause means that the routine body returns a SMALLINT and that DB2 casts the value to INTEGER before passing it along to the statement where the function reference occurs:

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

In this case, the routine must be written to generate a SMALLINT, as defined later in this section. Note that the CAST FROM data type must be *castable* to the RETURNS data type, therefore, it is not possible to arbitrarily choose another data type.

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

- SMALLINT

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

  Example:

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

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

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

  If you invoke the SIMPLE function using INTEGER data, (`... SIMPLE(1)...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the

function was not found, and end-users of this function might not perceive the reason for the message. In the preceding example, 1 is an INTEGER, so you can either cast it to SMALLINT or define the parameter as INTEGER.

- INTEGER or INT

  Valid. Represent in C as `SQLUDF_INTEGER` or `sqlint32`. You must `#include sqludf.h` or `#include sqlsystm.h` to pick up this definition.

  Example:

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

- BIGINT

  Valid. Represent in C as `SQLUDF_BIGINT` or `sqlint64`.

  Example:

  ```
  sqlint64 *arg3;          /* example for INTEGER */
  ```

  DB2 defines the `sqlint64` C language type to overcome differences between definitions of the 64-bit signed integer in compilers and operating systems. You must `#include sqludf.h` or `#include sqlsystm.h` to pick up the definition.

- REAL or FLOAT(*n*) where 1 <= *n* <= 24

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

  Example:

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

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

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

  Example:

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

- DECIMAL(p,s) or NUMERIC(p,s)

  Not valid because there is no C language representation. If you want to pass a decimal value, you must define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type. In the case of DOUBLE, you do not need to explicitly cast a decimal argument to a DOUBLE parameter, because DB2 promotes it automatically.

  Example:

  Suppose you have two columns, WAGE as DECIMAL(5,2) and HOURS as DECIMAL(4,1), and you wish to write a UDF to calculate weekly pay based on wage, number of hours worked and some other factors. The UDF could be as follows:

  ```
  CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
         RETURNS DECIMAL(7,2) CAST FROM DOUBLE
         ...;
  ```

  For the preceding UDF, the first two parameters correspond to the wage and number of hours. You invoke the UDF WEEKLY_PAY in your SQL select statement as follows:

  ```
  SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
  ```

  Note that no explicit casting is required because the DECIMAL arguments are castable to DOUBLE.

  Alternatively, you could define WEEKLY_PAY with CHAR arguments as follows:

  ```
  CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
         RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
         ...;
  ```

  You would invoke it as follows:

  ```
  SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
  ```

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

An advantage of using floating point parameters is that it is easy to perform arithmetic on the values in the routine; an advantage of using character parameters is that it is always possible to exactly represent the decimal value. This is not always possible with floating point.

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

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

  Example:

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

  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 DB2 does not promote VARCHAR arguments to CHAR and string literals are automatically considered as VARCHARs. For example, suppose you define a UDF as follows:

```
CREATE FUNCTION SIMPLE(INT,CHAR(1))...
```

  If you invoke the SIMPLE function using VARCHAR data, (`... SIMPLE(1,'A')...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not perceive the reason for the message. In the preceding example, `'A'` is VARCHAR, so you can either cast it to CHAR or define the parameter as VARCHAR.

- VARCHAR(n) FOR BIT DATA or LONG VARCHAR with or without the FOR BIT DATA modifier.

Valid. Represent VARCHAR(n) FOR BIT DATA in C as SQLUDF_VARCHAR_FBD. Represent LONG VARCHAR in C as SQLUDF_LONG. Otherwise represent these two SQL types in C as a structure similar to the following from the sqludf.h include file:

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

The [1] indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

These values are not represented as C null-terminated strings because the null-character could legitimately be part of the data value. The length is explicitly passed to the routine for parameters using the structure variable length. For the RETURNS clause, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable length, is the actual length of the data value.

Example:

```
struct sqludf_vc_fbd *arg1;  /* example for VARCHAR(n) FOR BIT DATA */
struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
```

- VARCHAR(n) without FOR BIT DATA.

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

  For a VARCHAR(n) parameter, DB2 will put a null in the (k+1) position, where k is the length of the particular string. The C string-handling functions are well suited for manipulation of these values. For a RETURNS VARCHAR(n) value or an output parameter of a stored procedure, the routine body must delimit the actual value with a null because DB2 will determine the result length from this null character.

  Example:

  ```
  char     arg2[51];      /* example for VARCHAR(50)   */
  char    *result;        /* also acceptable */
  ```

- DATE

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

  ```
  yyyy-mm-dd
  ```

  Example:

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

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

- TIME

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

  ```
  hh.mm.ss
  ```

  Example:

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

- TIMESTAMP

  Valid. Represent in C as SQLUDF_STAMP or CHAR(26), that is, as char...[27]. The timestamp value is always passed with format:

  ```
  yyyy-mm-dd-hh.mm.ss.nnnnnn
  ```

  Example:

  ```
  char     arg1[27];      /* example for TIMESTAMP  */
  char    *result;        /* also acceptable */
  ```

- GRAPHIC(n)

  Valid. Represent in C as SQLUDF_GRAPH or sqldbchar[n+1]. (This is a null-terminated graphic string). Note that you can use wchar_t[n+1] on operating systems where wchar_t is defined to be 2 bytes in length; however, sqldbchar is recommended.

  For a GRAPHIC(n) parameter, DB2 moves *n* double-byte characters to the buffer and sets the following two bytes to null. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For a RETURNS GRAPHIC(*n*) value or an output parameter of a stored procedure, DB2 looks for an embedded GRAPHIC null CHAR, and if it finds it, pads the value out to *n* with GRAPHIC blank characters.

  When defining graphic routine parameters, consider using VARGRAPHIC rather than GRAPHIC as DB2 does not promote VARGRAPHIC arguments to GRAPHIC. For example, suppose you define a routine as follows:

  ```
  CREATE FUNCTION SIMPLE(GRAPHIC)...
  ```

  If you invoke the SIMPLE function using VARGRAPHIC data, (... SIMPLE('*graphic_literal*')...), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not understand the reason for this message. In the preceding example, *graphic_literal* is a literal DBCS string that is interpreted as VARGRAPHIC data, so you can either cast it to GRAPHIC or define the parameter as VARGRAPHIC.

  Example:

  ```
  sqldbchar   arg1[14];      /* example for GRAPHIC(13)   */
  sqldbchar  *arg1;          /* also acceptable */
  ```

- VARGRAPHIC(n)

  Valid. Represent in C as SQLUDF_GRAPH or sqldbchar[n+1]. (This is a null-terminated graphic string). Note that you can use wchar_t[n+1] on operating systems where wchar_t is defined to be 2 bytes in length; however, sqldbchar is recommended.

  For a VARGRAPHIC(*n*) parameter, DB2 will put a graphic null in the (k+1) position, where *k* is the length of the particular occurrence. A graphic null refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For a RETURNS VARGRAPHIC(*n*) value or an output parameter of a stored procedure, the routine body must delimit the actual value with a graphic null, because DB2 will determine the result length from this graphic null character.

  Example:

  ```
  sqldbchar   args[51],    /* example for VARGRAPHIC(50) */
  sqldbchar   *result,     /* also acceptable  */
  ```

- LONG VARGRAPHIC

Valid. Represent in C as SQLUDF_LONGVARG or a structure:

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

Note that in the preceding structure, you can use wchar_t in place of sqldbchar on operating systems where wchar_t is defined to be 2 bytes in length, however, the use of sqldbchar is recommended.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed. Because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

These are not represented as null-terminated graphic strings. The length, in double-byte characters, is explicitly passed to the routine for parameters using the structure variable length. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable length, is the actual length of the data value, in double byte characters.

Example:

```
struct sqludf_vg *arg1;  /* example for VARGRAPHIC(n)   */
struct sqludf_vg *result; /* also for LONG VARGRAPHIC   */
```

- BLOB(n) and CLOB(n)

  Valid. Represent in C as SQLUDF_BLOB, SQLUDF_CLOB, or a structure:

  ```
  struct sqludf_lob
  {
      sqluint32     length;      /* length in bytes */
      char          data[1];      /* first byte of lob */
  };
  ```

  The [1] merely indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

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

  Example:

  ```
  struct sqludf_lob *arg1;  /* example for BLOB(n), CLOB(n) */
  struct sqludf_lob *result;
  ```

- DBCLOB(n)

  Valid. Represent in C as SQLUDF_DBCLOB or a structure:

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

  Note that in the preceding structure, you can use wchar_t in place of sqldbchar on operating systems where wchar_t is defined to be 2 bytes in length, however, the use of sqldbchar is recommended.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

These are not represented as null-terminated graphic strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable `length`, is the actual length of the data value, with all of these lengths expressed in double byte characters.

Example:

```
struct sqludf_lob *arg1;  /* example for DBCLOB(n) */
struct sqludf_lob *result;
```

- Distinct Types

  Valid or invalid depending on the base type. Distinct types will be passed to the UDF in the format of the base type of the UDT, so can be specified if and only if the base type is valid.

  Example:

```
struct sqludf_lob *arg1;  /* for distinct type based on BLOB(n) */
double            *arg2;  /* for distinct type based on DOUBLE  */
char              res[5]; /* for distinct type based on CHAR(4) */
```

- 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

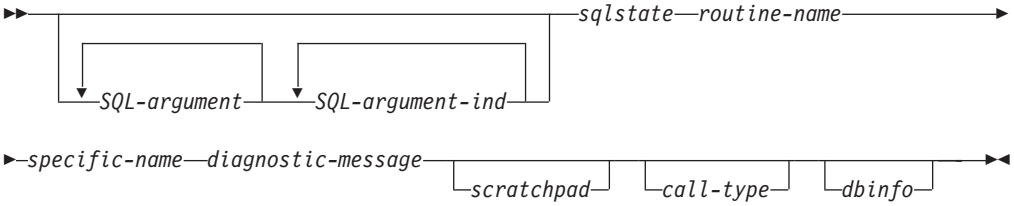
  Valid for parameters and results of UDFs and methods where an appropriate transform function exists. Structured type parameters will be passed to the function or method in the result type of the FROM SQL transform function. Structured type results will be passed in the parameter type of the TO SQL transform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

Passing arguments to C, C++, OLE, or COBOL routines

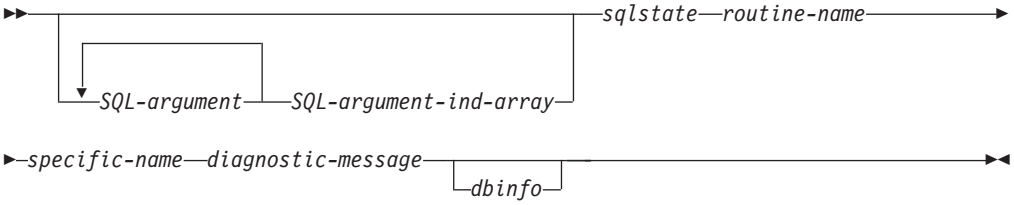
In addition to the SQL arguments that are specified in the DML reference for a routine, DB2 passes additional arguments to the external routine body. The nature and order of these arguments is determined by the parameter style with which you registered your routine. To ensure that information is exchanged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

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

PARAMETER STYLE SQL routines

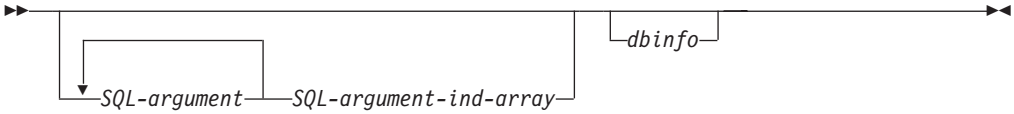


PARAMETER STYLE DB2SQL procedures



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 above parameter styles are described as follows:

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

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

Each *SQL-argument* is used as follows:

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

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

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

  For scalar functions and methods, the result data type is defined in the CAST FROM clause, if it is present, or in the RETURNS clause, if no CAST FROM clause is present.

  For table functions, DB2 defines a performance optimization where every defined column does not have to be returned to DB2. If you write

your UDF to take advantage of this feature, it returns only the columns required by the statement referencing the table function. For example, consider a CREATE FUNCTION statement for a table function defined with 100 result columns. If a given statement referencing the function is only interested in two of them, this optimization enables the UDF to return only those two columns for each row and not spend time on the other 98 columns. See the dbinfo argument below for more information on this optimization.

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

- INOUT parameter of a stored procedure

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

DB2 aligns the data for *SQL-argument* according to the data type and the server operating system, also known as platform.

SQL-argument-ind...

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

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

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

  This argument is set by DB2 before calling the routine. It contains one of the following values:

  0 The argument is present and not NULL.

  -1 The argument is present and its value is NULL.

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

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

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

  0 The result is not NULL.

  -1 The result is the NULL value.

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

For scalar functions and methods, DB2 treats a NULL result as an arithmetic error if the following is true:

– The database configuration parameter *dft_sqlmathwarn* is YES

– One of the input arguments is a null because of an arithmetic error

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

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

- INOUT parameter of a stored procedure

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

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

SQL-argument-ind-array

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

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

- IN parameter of a stored procedure

  This element is set by DB2 before calling the routine. It contains one of the following values:

  0 The argument is present and not NULL.

  -1 The argument is present and its value is NULL.

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

- OUT parameter of a stored procedure

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

  0 or positive
  The result is not NULL.

  negative
  The result is the NULL value.

- INOUT parameter of a stored procedure

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

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

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

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

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

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

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

```
schema.routine
```

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

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

This form enables you to use the same routine body for multiple external routines, and still differentiate between the routines when it is invoked.

Note: Although it is possible to include the period in object names and schema names, it is not recommended. For example, if a function, `ROTATE` is in a schema, `OBJ.OP`, the routine name that is passed to the function is `OBJ.OP.ROTATE`, and it is not obvious if the schema name is `OBJ` or `OBJ.OP`.

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

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

Two examples are:

```
WILLIE_FIND_FEB99   SQL9904281052440430
```

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

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

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

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

When the routine returns either an error or a warning, using the *sqlstate* argument described previously, it can include descriptive information here. DB2 includes this information as a token in its message.

DB2 sets the first character to null before calling the routine. Upon return, it treats the string as a C null-terminated string. This string will be included in the SQLCA as a token for the error condition. At least the first part of this string will appear in the SQLCA or DB2 CLP message. However, the actual number of characters that will appear depends on the lengths of the other tokens, because DB2 truncates the tokens to conform to the limit on total token length imposed by the SQLCA. Avoid using X'FF' in the text since this character is used to delimit tokens in the SQLCA.

The routine should not return more text than will fit in the VARCHAR(70) buffer that is passed to it. An overwrite by the routine can cause unpredictable results or an abend.

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

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

scratchpad

This argument is set by DB2 before invoking the UDF or method. It is only present for functions and methods that specified the SCRATCHPAD keyword during registration. This argument is a structure, exactly like the structure used to pass a value of any of the LOB data types, with the following elements:

- An INTEGER containing the length of the scratchpad. Changing the length of the scratchpad will result in SQLCODE -450 (SQLSTATE 39501)
- The actual scratchpad initialized to all binary 0s as follows:
  - For scalar functions and methods, it is initialized before the first call, and not generally looked at or modified by DB2 thereafter.
  - For table functions, the scratchpad is initialized prior to the FIRST call to the UDF if FINAL CALL is specified on the CREATE FUNCTION. After this call, the scratchpad content is totally under control of the table function. If NO FINAL CALL was specified or defaulted for a table function, then the scratchpad is initialized for each OPEN call, and the scratchpad content is completely under control of the table function between OPEN calls. (This can be very important for a table function used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH and CLOSE calls, the table function will also receive FIRST and FINAL calls, for the purpose of scratchpad maintenance and resource release.)

The scratchpad can be mapped in your routine using the same type as either a CLOB or a BLOB, since the argument passed has the same structure.

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

If a scalar UDF or method that uses a scratchpad is referenced in a subquery, DB2 might decide to refresh the scratchpad between invocations of the subquery. This refresh occurs after a final-call is made, if FINAL CALL is specified for the UDF.

DB2 initializes the scratchpad so that the data field is aligned for the storage of any data type. This can result in the entire scratchpad structure, including the length field, being improperly aligned.

call-type

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

All the current possible values for *call-type* follow. Your UDF or method should contain a switch or case statement that explicitly tests for all the expected values, rather than containing "if A do AA, else if B do BB, else it must be C so do CC" type logic. This is because it is possible that additional call types will be added in the future, and if you do not explicitly test for condition C you will have trouble when new possibilities are added.

Note:

1. For all values of *call-type*, it might be appropriate for the routine to set a *sqlstate* and *diagnostic-message* return value. This information will not be repeated in the following descriptions of each *call-type*. For all calls DB2 will take the indicated action as described previously for these arguments.
2. The include file `sqludf.h` is intended for use with routines. The file contains symbolic defines for the following *call-type* values, which are spelled out as constants.

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

SQLUDF_FIRST_CALL (-1)

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

Note: If SCRATCHPAD is specified but FINAL CALL is not, then the routine will not have this *call-type* argument to identify the very first call. Instead, it will have to rely on the all-zero state of the scratchpad.

SQLUDF_NORMAL_CALL (0)

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

SQLUDF_FINAL_CALL (1)

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

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

Releasing resources

A scalar UDF or method is expected to release resources it has required, for example, memory. If FINAL CALL is specified for the routine, then that FINAL call is a natural place to release resources, provided that SCRATCHPAD is also specified and is used to track the resource. If FINAL CALL is not specified, then any resource acquired should be released on the same call.

For table functions *call-type* contains:

SQLUDF_TF_FIRST (-2)
This is the FIRST call, which only occurs if the FINAL CALL keyword was specified for the UDF. The *scratchpad* is set to binary zeros before this call. Argument values are passed to the table function. The table function can acquire memory or perform other one-time only resource initialization. This is not an OPEN call, that call follows this one. On a FIRST call the table function should not return any data to DB2 as DB2 ignores the data.

SQLUDF_TF_OPEN (-1)
This is the OPEN call. The *scratchpad* will be initialized if NO FINAL CALL is specified, but not necessarily otherwise. All SQL argument values are passed to the table function on OPEN. The table function should not return any data to DB2 on the OPEN call.

SQLUDF_TF_FETCH (0)
This is a FETCH call, and DB2 expects the table function to return either a row comprising the set of return values, or an end-of-table condition indicated by SQLSTATE value '02000'. If *scratchpad* is passed to the UDF, then on entry it is untouched from the previous call.

SQLUDF_TF_CLOSE (1)
This is a CLOSE call to the table function. It balances the OPEN call, and can be used to perform any external CLOSE processing (for example, closing a source file), and resource release (particularly for the NO FINAL CALL case).

In cases involving a join or a subquery, the OPEN/FETCH.../ CLOSE call sequences can repeat within the execution of a statement, but there is only one FIRST call and only one FINAL call. The FIRST and FINAL call only occur if FINAL CALL is specified for the table function.

SQLUDF_TF_FINAL (2)
This is a FINAL call, which only occurs if FINAL CALL was

specified for the table function. It balances the FIRST call, and occurs only once per execution of the statement. It is intended for the purpose of releasing resources.

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

Releasing resources

Write routines to release any resources that they acquire. For table functions, there are two natural places for this release: the CLOSE call and the FINAL call. The CLOSE call balances each OPEN call and can occur multiple times in the execution of a statement. The FINAL call only occurs if FINAL CALL is specified for the UDF, and occurs only once per statement.

If you can apply a resource across all OPEN/FETCH/CLOSE sequences of the UDF, write the UDF to acquire the resource on the FIRST call and free it on the FINAL call. The scratchpad is a natural place to track this resource. For table functions, if FINAL CALL is specified, the scratchpad is initialized only before the FIRST call. If FINAL CALL is not specified, then it is reinitialized before each OPEN call.

If a resource is specific to each OPEN/FETCH/CLOSE sequence, write the UDF to free the resource on the CLOSE call.

Note: When a table function is in a subquery or join, it is very possible that there will be multiple occurrences of the OPEN/FETCH/CLOSE sequence, depending on how the DB2 Optimizer chooses to organize the execution of the statement.

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

dbinfo
This argument is set by DB2 before calling the routine. It is only present if the CREATE statement for the routine specifies the DBINFO keyword. The argument is the `sqludf_dbinfo` structure defined in the header file `sqludf.h`. The variables in this structure that contain names and identifiers might be longer than the longest value possible in this release of DB2, but they are defined this way for compatibility with future releases. You can use the length variable that complements each name and identifier variable to read or extract the portion of the variable that is actually used. The *dbinfo* structure contains the following elements:

1. Database name length (dbnamelen)

   The length of *database name* below. This field is an unsigned short integer.
2. Database name (dbname)

The name of the currently connected database. This field is a long identifier of 128 characters. The *database name length* field described previously identifies the actual length of this field. It does not contain a null terminator or any padding.

3. Application Authorization ID Length (authidlen)

   The length of *application authorization ID* below. This field is an unsigned short integer.

4. Application authorization ID (authid)

   The application run-time authorization ID. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *application authorization ID length* field described above identifies the actual length of this field.

5. Environment code pages (codepg)

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

   The cdpg_db2 structure is made up of an array (db2_ccsids_triplet) of three sets of code page information representing the possible encoding schemes in the database as follows:

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

   Following the encoding scheme information is the array index of the encoding scheme for the routine (db2_encoding_scheme).Each element of the array is composed of three fields:

   - db2_sbcs. Single byte code page, an unsigned long integer.
   - db2_dbcs. Double byte code page, an unsigned long integer.
   - db2_mixed. Composite code page (also called mixed code page), an unsigned long integer.

6. Schema name length (tbschemalen)

   The length of *schema name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

7. Schema name (tbschema)

   Schema for the *table name* below. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *schema name length* field described previously identifies the actual length of this field.

8. Table name length (tbnamelen)

   The length of the *table name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

9. Table name (tbname)

   This is the name of the table being updated or inserted. This field is set only if the routine reference is the right-side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT

statement. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *table name length* field described previously, identifies the actual length of this field. The *schema name* field described previously, together with this field form the fully qualified table name.

10. Column name length (colnamelen)

    Length of *column name* below. It contains a 0 (zero) if a column name is not passed. This field is an unsigned short integer.

11. Column name (colname)

    Under the exact same conditions as for table name, this field contains the name of the column being updated or inserted; otherwise, it is not predictable. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *column name length* field described above, identifies the actual length of this field.

12. Version/Release number (ver_rel)An 8 character field that identifies the product and its version, release, and modification level with the format *pppvvrrm* where:

    - *ppp* identifies the product as follows:

      DSN DB2 for z/OS or OS/390

      ARI SQL/DS or DB2 for VM or VSE

      QSQ DB2 Database for System i

      SQL DB2 Database for Linux, UNIX, and Windows
    - *vv* is a two digit version identifier.
    - *rr* is a two digit release identifier.
    - *m* is a one digit modification level identifier.

13. Reserved field (resd0)

    This field is for future use.

14. Platform (platform)

    The operating system (platform) for the application server, as follows:

    SQLUDF_PLATFORM_AIX
    AIX®

    SQLUDF_PLATFORM_HP
    HP-UX

    SQLUDF_PLATFORM_LINUX
    Linux

    SQLUDF_PLATFORM_MVS
    OS/390

    SQLUDF_PLATFORM_NT
    Windows 2000, Windows XP

    SQLUDF_PLATFORM_SUN
    Solaris operating system

    SQLUDF_PLATFORM_WINDOWS95
    Windows 95, Windows 98, Windows Me

    SQLUDF_PLATFORM_UNKNOWN
    Unknown operating system or platform

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

15. Number of table function column list entries (numtfcol)

    The number of non-zero entries in the table function column list specified in the *table function column list* field below.

16. Reserved field (resd1)

    This field is for future use.

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

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

18. Reserved field (resd2)

    This field is for future use.

19. Table function column list (tfcolumn)

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

    This field is used only for table functions. Only the first *n* entries, where *n* is specified in the *number of table function column list entries* field, `numtfcol`, are of interest. *n* can be equal to 0, and *n* is less than or equal to the number of result columns defined for the function in the RETURNS TABLE(...) clause of the CREATE FUNCTION statement. The values correspond to the ordinal numbers of the columns that this statement needs from the table function. A value of '1' means the first defined result column, '2' means the second defined result column, and so on, and the values can be in any order. Note that *n* could be equal to zero, that is, the variable `numtfcol` might be zero, for a statement similar to `SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ`, where no actual column values are needed by the query.

    This array represents an opportunity for optimization. The UDF need not return all values for all the result columns of the table function, only those needed in the particular context, and these are the columns identified (by number) in the array. Since this optimization can complicate the UDF logic in order to gain the performance benefit, the UDF can choose to return every defined column.

20. Unique application identifier (appl_id)

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

    The string has a maximum length of 32 characters, and its exact format depends on the type of connection established between the client and DB2. Generally it takes the form:

    ```
    x.y.ts
    ```

    where the *x* and *y* vary by connection type, but the *ts* is a 12 character time stamp of the form YYMMDDHHMMSS, which is potentially adjusted by DB2 to ensure uniqueness.

    ```
    Example:  *LOCAL.db2inst.980707130144
    ```

21. Reserved field (resd3)

    This field is for future use.

Graphic host variables in C and C++ routines

Any routine written in C or C++ that receives or returns graphic data through its parameter input or output should generally be precompiled with the WCHARTYPE NOCONVERT option. This is because graphic data passed through these parameters is considered to be in DBCS format, rather than the `wchar_t` process code format. Using NOCONVERT means that graphic data manipulated in SQL statements in the routine will also be in DBCS format, matching the format of the parameter data.

With WCHARTYPE NOCONVERT, no character code conversion occurs between the graphic host variable and the database manager. The data in a graphic host variable is sent to, and received from, the database manager as unaltered DBCS characters. If you do not use WCHARTYPE NOCONVERT, it is still possible for you to manipulate graphic data in `wchar_t` format in a routine; however, you must perform the input and output conversions manually.

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

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

C++ type decoration

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

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

and

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

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

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

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

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

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

On UNIX operating systems, the type-decorated function name can be determined from the `.o` (object) file, or from the shared library, using the `nm` command. This command can produce considerable output, so it is suggested that you pipe the output through `grep` to look for the right line, as follows:

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

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

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

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

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

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

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

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

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

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

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

In this example, the UDFs fold and findvwl are not type-decorated by the compiler, and should be registered in the CREATE FUNCTION statement using

their plain names. Similarly, if a C++ stored procedure or method is coded with `extern "C"`, its undecorated function name would be used in the CREATE statement.

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 follow the steps in the procedure below for returning results from a C or C++ procedure.

Creating 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 DB2 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, pre-compilation, compilation and linking of code, and the deployment of source code.

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++.

Prerequisites

- Knowledge of C and C++ routine implementation. To learn about C and C++ routines in general see:
  - "C and C++ routines" on page 204
- The DB2 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 DB2 supported C or C++ compiler for routine development.
- 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.

Procedure

1. Code the routine logic in the chosen programming language: C or C++.
   - For general information about C and C++ routines and C and C++ routine features, see the topics referenced in the Prerequisites section.
   - Include any C or C++ header files required for additional C functionality as well as the DB2 C or C++ header files required for SQL data type and SQL execution support. Include the following header files: sqludf.h, sql.h, sqlda.h, sqlca.h, and memory.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 208
  - "Parameter style SQL C and C++ procedures" on page 210
  - "Parameter style SQL C and C++ functions" on page 213
- 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 DB2 SQL data types. For more on data type mapping between DB2 and C or C++ data types refer to:
  - "Supported SQL data types in C and C++ routines" on page 220
- 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 244
- 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 247
3. Copy the library into the DB2 *function directory* on the database server. It is recommended that you store libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the 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 DB2 will look 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.

To invoke your C or C++ routine, see Chapter 11, "Routine invocation," on page 333.

Building C and C++ routine code

Once embedded SQL C or C++ routine implementation code has been written, it must be built into a library and deployed before the routine can be invoked. Although the steps required to build embedded SQL C and C++ routines are similar to those required to build embedded SQL C and C++ applications, there are some differences. The same steps can be followed if there are no embedded SQL statements within the routines - the procedure will be faster and simpler.

There are two ways to build C and C++ routines:

- Using DB2 sample build scripts (UNIX) or build batch files (Windows)
- Entering DB2 and C or C++ compiler commands from a DB2 Command Window

The DB2 sample build scripts and batch files for routines are designed for building DB2 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 DB2 sample build scripts and batch files for C and C++. In general it is easiest to build embedded SQL routines using the build scripts or batch files, which can easily be modified if required, however it is often helpful to know how to build routines from DB2 Command Window as well.

For more information on each of the methods for building routines, refer to the related links.

Building C and C++ routine code using sample bldrtn scripts

Building C and C++ routine source code is a sub-task of creating C and C++ routines. This task can be done quickly and easily using DB2 sample build scripts (UNIX) and batch files (Windows). The sample build scripts can be used for source code with or without embedded SQL statements. The build scripts take care of the pre-compilation, compilation, and linking of C and C++ source code that would otherwise have to be done in individual steps from the command line. They also take care of binding any packages to the specified database.

The sample build scripts for building C and C++ routines are named `bldrtn`. They are located in DB2 directories along with sample programs that can be built with them as follows:

- For C: `sqllib/samples/c/`
- For C++: `sqllib/samples/cpp/`

The `bldrtn` script can be used to build a source code file containing both procedures and function implementations. The script does the following:

- 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 DB2 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 will be established

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

Prerequisites

- Source code file containing one or more routine implementations.
- The name of the database within the current DB2 instance in which the routines are to be created.

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

1. Open a 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 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 file containing the routine implementations was built, stop and restart the database to ensure the new version of the shared library is used by DB2. 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. After routine creation is completed you will be able to invoke your routines.

Building routines in C or C++ using the sample build script (UNIX)

DB2 provides build scripts for compiling and linking C and C++ programs. These are located in the `sqllib/samples/c` directory for routines in C and `sqllib/samples/cpp` directory for routines in C++, along with sample programs that can be built with these files.

The script, bldrtn, contains the commands to build routines (stored procedures and user-defined functions). The script compiles the routines into a shared library that can be loaded by the database manager and called by a client application.

The first parameter, $1, specifies the name of your source file. The second parameter, $2, specifies the name of the database to which you want to connect.

The database parameter is optional. If no database name is supplied, the program uses the default sample database. And since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

The following examples show you how to build routine shared libraries with:

- stored procedures
- non-embedded SQL user-defined functions (UDFs)
- embedded SQL user-defined functions (UDFs)

Stored procedure shared library

To build the sample program spserver from the source file spserver.sqc for C and spserver.sqC for C++:

1. If connecting to the sample database, enter the build script name and program name:

   ```
   bldrtn spserver
   ```

   If connecting to another database, also enter the database name:

   ```
   bldrtn spserver database
   ```

   The script copies the shared library to the server in the path sqllib/function.
2. Next, catalog the routines by running the spcat script on the server:

   ```
   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 call the spdrop.db2 and spcreate.db2 scripts individually.
3. Then, if this is not the first time the stored procedure is built, stop and restart the database to ensure the new version of the shared library is recognized. You can do this by entering db2stop followed by db2start on the command line.

Once you build the shared library, spserver, you can build the client application, spclient, that accesses the shared library.

You can build spclient by using the script, bldapp.

To call the stored procedures in the shared library, run the sample client application by entering: spclient *database userid password*

where

database
Is the name of the database to which you want to connect. The name could be sample, or its alias, or another database name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the shared library, spserver, and executes a number of stored procedure functions on the server database. The output is returned to the client application.

Embedded SQL UDF shared library

To build the embedded SQL user-defined function program, udfemsrv, from the source file udfemsrv.sqc for C and udfemsrv.sqC for C++, if connecting to the sample database, enter the build script name and program name:

```
bldrtn udfemsrv
```

If connecting to another database, also enter the database name:

```
bldrtn udfemsrv database
```

The script copies the UDF to the sqllib/function directory.

Once you build udfemsrv, you can build the client application, udfemcli, that calls it. You can build the udfemcli client program from the source file udfemcli.sqc, in sqllib/samples/c, using the script, bldapp.

To call the UDFs in the shared library, run the client application by entering:
udfemcli *database userid password*

where

database
Is the name of the database to which you want to connect. The name could be sample, or its alias, or another database name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the shared library, udfemsrv, and executes the user-defined functions on the server database. The output is returned to the client application.

Building C/C++ routines on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in C and C++. These are located in the sqllib\samples\c and sqllib\samples\cpp directories, along with sample programs that can be built with these files.

The batch file bldrtn.bat contains the commands to build embedded SQL routines (stored procedures and user-defined functions). The batch file builds a DLL on the server. It takes two parameters, represented inside the batch file by the variables %1 and %2.

The first parameter, %1, specifies the name of your source file. The batch file uses the source file name for the DLL name. The second parameter, %2, specifies the

name of the database to which you want to connect. Since the DLL must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, the source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

The following examples show you how to build routine DLLs with:

- stored procedures
- non-embedded SQL user-defined functions (UDFs)
- embedded SQL user-defined functions (UDFs)

Stored procedure DLL

To build the `spserver` DLL from either the C source file, `spserver.sqc`, or the C++ source file, `spserver.sqx`:

1. Enter the batch file name and program name:

   ```
   bldrtn spserver
   ```

   If connecting to another database, also enter the database name:

   ```
   bldrtn spserver database
   ```

   The batch file uses the module definition file `spserver.def`, contained in the same directory as the sample programs, to build the DLL. The batch file copies the DLL, `spserver.dll`, to the server in the path `sqllib\function`.
2. Next, catalog the routines by running the `spcat` script on the server:

   ```
   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 call the `spdrop.db2` and `spcreate.db2` scripts individually.
3. Then, stop and restart the database to allow the new DLL to be recognized. If necessary, set the file mode for the DLL so the DB2 instance can access it.

Once you build the DLL, `spserver`, you can build the client application `spclient` that calls it.

You can build `spclient` by using the batch file, `bldapp.bat`.

To call the DLL, run the sample client application by entering:

`spclient` *database userid password*

where

database
Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another database name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the DLL, `spserver`, and executes a number of routines on the server database. The output is returned to the client application.

Non-embedded SQL UDF DLL

To build the user-defined function udfsrv from the source file udfsrv.c, enter:

```
bldrtn udfsrv
```

The batch file uses the module definition file, udfsrv.def, contained in the same directory as the sample program files, to build the user-defined function DLL. The batch file copies the user-defined function DLL, udfsrv.dll, to the server in the path sqllib\function.

Once you build udfsrv, you can build the client application, udfcli, that calls it. DB2 CLI, as well as embedded SQL C and C++ versions of this program are provided.

You can build the DB2 CLI udfcli program from the udfcli.c source file in sqllib\samples\cli using the batch file bldapp.

You can build the embedded SQL C udfcli program from the udfcli.sqc source file in sqllib\samples\c using the batch file bldapp.

You can build the embedded SQL C++ udfcli program from the udfcli.sqx source file in sqllib\samples\cpp using the batch file bldapp.

To run the UDF, enter:

```
udfcli
```

The calling application calls the ScalarUDF function from the udfsrv DLL.

Embedded SQL UDF DLL

To build the embedded SQL user-defined function library udfemsrv from the C source file udfemsrv.sqc in sqllib\samples\c, or from the C++ source file udfemsrv.sqx in sqllib\samples\cpp, enter:

```
bldrtn udfemsrv
```

If connecting to another database, also enter the database name:

```
bldrtn udfemsrv database
```

The batch file uses the module definition file, udfemsrv.def, contained in the same directory as the sample programs, to build the user-defined function DLL. The batch file copies the user-defined function DLL, udfemsrv.dll, to the server in the path sqllib\function.

Once you build udfemsrv, you can build the client application, udfemcli, that calls it. You can build udfemcli from the C source file udfemcli.sqc in sqllib\samples\c, or from the C++ source file udfemcli.sqx in sqllib\samples\cpp using the batch file bldapp.

To run the UDF, enter:

```
udfemcli
```

The calling application calls the UDFs in the udfemsrv DLL.

Building C and C++ routine code from DB2 Command Window

Building C and C++ routine source code is a sub-task of creating C and C++ routines. This task can be done manually from the command line. The same procedure can be followed regardless of whether there are embedded SQL statements within the C or C++ routine code or not. The task steps include pre-compilation, compilation, and linking of C and C++ source code containing routine implementations, binding the generated package (if there were embedded SQL statements), and deploying the routine library. You might choose to do this task from a DB2 Command Window as part of testing the use of a precompiler, compiler, or bind option, if you want to defer binding the routine packages until a later time, or if you are developing customized build scripts.

As an alternative, you can use DB2 sample build scripts to simplify this task. Refer to: Building embedded SQL C and C++ routine code using sample build scripts.

Prerequisites

- Source code file containing one or more embedded SQL C or C++ routine implementations.
- The name of the database within the current DB2 instance in which the routines are to be created.
- The operating specific compile and link options required for building C and C++ routines. Refer to the topics referenced in the related links at the bottom of this topic.

To build a source code file that contains one or more routine code implementations, follow the steps below. An example follows that demonstrates each of the steps:

1. Open a 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 will be 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. This requires the use of some DB2 specific compile and link options for the compiler being used.
8. Copy the shared library to the DB2 function directory on the database server.
9. 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 DB2. You can do this by issuing the db2stop command followed by the db2start command.

Once you have successfully built and deployed the routine library, you should complete the steps associated with the task of creating C and C++ routines. Creating C and C++ routines includes a step for executing the CREATE statement for each routine that was implemented in the source code file. This step must also be completed before you will be able to invoke the routines.

Example

The following example demonstrates the re-building of an embedded SQL C++ source code file named myfile.sqC containing routine implementations. The

routines are being built on an AIX operating system using the default supported IBM VisualAge® C++ compiler to generate a 32-bit routine library.

1. Open a 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 will be created.

   ```
   db2 connect to <database-name>
   ```

4. Precompile the source code file using the PREPARE command.

   ```
   db2 prep myfile.sqC bindfile
   ```

   The precompiler will generate output indicating if the precompilation proceeded successfully or if there were any errors. This step generates bindfile named myfile.bnd that can be used to generate a package in the next step.
5. Bind the package that was generated to the database using the BIND command.

   ```
   db2 bind myfile.bnd
   ```

   The bind utility will generate output indicating if the bind proceeded successfully or if there were any errors.
6. Compile the source code file using the recommended compile and link options:

   ```
   xlC_r -qstaticinline -I$HOME/sqllib/include -c $myfile.C
   ```

   The compiler will generate output if there are any errors. This step generates an export file named myfile.exp.
7. Link the source code file to generate a shared library.

   ```
   xlC_r -qmkshrobj -o $1 $1.o -L$ HOME/sqllib/include/lib32 -lDB2
   ```

   The linker will generate output if there are any errors. This step generates a shared library file name myfile.
8. Copy the shared library to the DB2 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 DB2 looks for routine libraries. Refer to the topic on creating C and C++ routines for more on deploying routine libraries.
9. Stop and restart the database as this is a re-building of a previously built routine source code file.

   ```
   db2stop
   db2start
   ```

Building C and C++ routines is generally most easily done using the operating specific sample build scripts which also can be used as a reference for how to build routines from the command line.

Compile and link options for C and C++ routines

AIX C routine compile and link options

The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the AIX IBM C compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

xlc_r Use the multi-threaded version of the IBM C compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

$EXTRA_CFLAG
Contains "-q64" for an instance where 64-bit support is enabled; otherwise, it contains no value.

-I$DB2PATH/include
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`.

-c Perform compile only; no link. Compile and link are separate steps.

Link options:

xlc_r Use the multi-threaded version of the compiler as a front end for the linker.

$EXTRA_CFLAG
Contains "-q64" for an instance where 64-bit support is enabled; otherwise, it contains no value.

-qmkshrobj
Create the shared library.

-o $1 Specify the output file name.

$1.o Specify the object file.

-ldb2 Link with the DB2 library.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`.

-bE:$1.exp
Specify an export file. The export file contains a list of the routines.

Refer to your compiler documentation for additional compiler options.

AIX C++ routine compile and link options

The following are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the AIX IBM XL C/C++ compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

xlC_r The multi-threaded version of the IBM XL C/C++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

$EXTRA_CFLAG
Contains "-q64" for an instance where 64-bit support is enabled; otherwise, it contains no value.

-I$DB2PATH/include
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`.

-c Perform compile only; no link. Compile and link are separate steps.

Link options:

xlC_r Use the multi-threaded version of the compiler as a front-end for the linker.

$EXTRA_CFLAG
Contains "-q64" for an instance where 64-bit support is enabled; otherwise, it contains no value.

-qmkshrobj
Create a shared library.

-o $1 Specify the output as a shared library file.

$1.o Specify the program object file.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`.

-ldb2 Link with the DB2 library.

-bE:$1.exp
Specify an export file. The export file contains a list of the routines.

Refer to your compiler documentation for additional compiler options.

HP-UX C routine compile and link options

The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the HP-UX C compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

cc The C compiler.

$EXTRA_CFLAG
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.

+DD64 Must be used to generate 64-bit code for HP-UX on IA64.

+DD32 Must be used to generate 32-bit code for HP-UX on IA64.

+DA2.0W
Must be used to generate 64-bit code for HP-UX on PA-RISC.

+DA2.0N
Must be used to generate 32-bit code for HP-UX on PA-RISC.

+u1 Allow unaligned data access. Use only if your application uses unaligned data.

+z Generate position-independent code.

-Ae Enables HP ANSI extended mode.

-I$DB2PATH/include
Specify the location of the DB2 include files. For example: `-I$DB2PATH/include`.

-D_POSIX_C_SOURCE=199506L
POSIX thread library option that ensures _REENTRANT is defined, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

-c Perform compile only; no link. Compile and link are separate steps.

Link options:

ld Use the linker to link.

-b Create a shared library rather than a normal executable.

-o $1 Specify the output as a shared library file.

$1.o Specify the program object file.

$EXTRA_LFLAG
Specify the runtime path. If set, for 32-bit it contains the value `"+b$HOME/sqllib/lib32"`, and for 64-bit: `"+b$HOME/sqllib/lib64"`. If not set, it contains no value.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For 32-bit: `$HOME/sqllib/lib32`; for 64-bit: `$HOME/sqllib/lib64`.

-ldb2 Link with the DB2 library.

-lpthread
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

HP-UX C++ routine compile and link options

The following are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the HP-UX C++ compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

aCC The HP aC++ compiler.

$EXTRA_CFLAG
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.

+DD64 Must be used to generate 64-bit code for HP-UX on IA64.

+DD32 Must be used to generate 32-bit code for HP-UX on IA64.

+DA2.0W
Must be used to generate 64-bit code for HP-UX on PA-RISC.

+DA2.0N
Must be used to generate 32-bit code for HP-UX on PA-RISC.

+u1 Allows unaligned data access.

+z Generate position-independent code.

-ext Allow various C++ extensions including "long long" support.

-mt Allows threads support for the HP aC++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

-I$DB2PATH/include
Specify the location of the DB2 include files. For example: `$DB2PATH/include`

-c Perform compile only; no link. Compile and link are separate steps.

Link options:

aCC Use the HP aC++ compiler as a front end for the linker.

$EXTRA_CFLAG
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.

+DD64 Must be used to generate 64-bit code for HP-UX on IA64.

+DD32 Must be used to generate 32-bit code for HP-UX on IA64.

+DA2.0W
Must be used to generate 64-bit code for HP-UX on PA-RISC.

+DA2.0N
Must be used to generate 32-bit code for HP-UX on PA-RISC.

-mt Allows threads support for the HP aC++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

-b Create a shared library rather than a normal executable.

-o $1 Specify the executable.

$1.o Specify the program object file.

$EXTRA_LFLAG
Specify the runtime path. If set, for 32-bit it contains the value `-Wl,+b$HOME/sqllib/lib32`, and for 64-bit: `-Wl,+b$HOME/sqllib/lib64`. If not set, it contains no value.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For 32-bit: `"$HOME/sqllib/lib32"`; for 64-bit: `"$HOME/sqllib/lib64"`.

-ldb2 Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

Linux C routine compile and link options

The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the Linux C compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

$CC The gcc or xlc_r compiler

$EXTRA_C_FLAGS
Contains one of the following:
- -m31 on Linux for zSeries only, to build a 32-bit library;
- -m32 on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- -m64 on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

-I$DB2PATH/include
Specify the location of the DB2 include files.

-c Perform compile only; no link. This script file has separate compile and link steps.

-D_REENTRANT
Defines _REENTRANT, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

Link options:

$CC The gcc or xlc_r compiler; use the compiler as a front end for the linker.

$LINK_FLAGS
Contains the value "$EXTRA_C_FLAGS $SHARED_LIB_FLAG"

$EXTRA_C_FLAGS
Contains one of the following:
- -m31 on Linux for zSeries only, to build a 32-bit library;
- -m32 on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- -m64 on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

$SHARED_LIB_FLAG
Contains -shared for gcc compiler or -qmkshrobj for xlc_r compiler.

-o $1 Specify the executable.

$1.o Include the program object file.

$EXTRA_LFLAG
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value "-Wl,-rpath,$DB2PATH/lib32". For 64-bit it contains the value "-Wl,-rpath,$DB2PATH/lib64".

-L$DB2PATH/$LIB
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: $HOME/sqllib/lib32, and for 64-bit: $HOME/sqllib/lib64.

-ldb2 Link with the DB2 library.

-lpthread
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

Linux C++ routine compile and link options

These are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the Linux C++ compiler, as demonstrated in the bldrtn build script.

Compile and link options for bldrtn

Compile options:

g++ The GNU/Linux C++ compiler.

$EXTRA_C_FLAGS
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

-fpic Generate position independent code.

-I$DB2PATH/include
Specify the location of the DB2 include files.

-c Perform compile only; no link. This script file has separate compile and link steps.

-D_REENTRANT
Defines _REENTRANT, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

Link options:

g++ Use the compiler as a front end for the linker.

$EXTRA_C_FLAGS
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

-shared
Generate a shared library.

-o $1 Specify the executable.

$1.o Include the program object file.

$EXTRA_LFLAG
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib32″. For 64-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib64″.

-L$DB2PATH/$LIB
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

-ldb2 Link with the DB2 library.

-lpthread
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

Solaris C routine compile and link options

These are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the Forte C compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

cc The C compiler.

-xarch=$CFLAG_ARCH
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either "v8plusa" for 32-bit, or "v9" for 64-bit.

-mt Allow multi-threaded support, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

-DUSE_UI_THREADS
Allows Sun's "UNIX International" threads APIs.

-Kpic Generate position-independent code for shared libraries.

-I$DB2PATH/include
Specify the location of the DB2 include files.

-c Perform compile only; no link. This script has separate compile and link steps.

Link options:

cc Use the compiler as a front end for the linker.

-xarch=$CFLAG_ARCH
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either "v8plusa" for 32-bit, or "v9" for 64-bit.

-mt This is required because the DB2 library is linked with `-mt`.

-G Generate a shared library.

-o $1 Specify the executable.

$1.o Include the program object file.

-L$DB2PATH/$LIB
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

$EXTRA_LFLAG
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value "-R$DB2PATH/lib32", and for 64-bit it contains the value "-R$DB2PATH/lib64".

-ldb2 Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

Solaris C++ routine compile and link options

These are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the Forte C++ compiler, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

CC The C++ compiler.

-xarch=$CFLAG_ARCH
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either "v8plusa" for 32-bit, or "v9" for 64-bit.

-mt Allow multi-threaded support, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

-DUSE_UI_THREADS
Allows Sun's "UNIX International" threads APIs.

-Kpic Generate position-independent code for shared libraries.

-I$DB2PATH/include
Specify the location of the DB2 include files.

-c Perform compile only; no link. This script has separate compile and link steps.

Link options:

CC Use the compiler as a front end for the linker.

-xarch=$CFLAG_ARCH
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either "v8plusa" for 32-bit, or "v9" for 64-bit.

-mt This is required because the DB2 library is linked with `-mt`.

-G Generate a shared library.

-o $1 Specify the executable.

$1.o Include the program object file.

-L$DB2PATH/$LIB
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

$EXTRA_LFLAG
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value "-R$DB2PATH/lib32", and for 64-bit it contains the value "-R$DB2PATH/lib64".

-ldb2 Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

Windows C and C++ routine compile and link options

The following are the compile and link options recommended by DB2 for building C and C++ routines (stored procedures and user-defined functions) on Windows with the Microsoft Visual C++ compiler, as demonstrated in the `bldrtn.bat` batch file.

Compile and link options for bldrtn

Compile options: **%BLDCOMP%** Variable for the compiler. The default is `cl`, the Microsoft Visual C++ compiler. It can be also set to `icl`, the Intel™ C++ Compiler for 32-bit and 64-bit applications, or `ecl`, the Intel C++ Compiler for Itanium 64-bit applications. **-Zi** Enable debugging information **-Od** Disable optimization. **-c** Perform compile only; no link. Compile and link are separate steps. **-W2** Output warning, error, and severe and unrecoverable error messages. **-DWIN32** Compiler option necessary for Windows operating systems. **-MD** Link using MSVCRT.LIB
Link options: **link** Use the linker to link. **-debug** Include debugging information. **-out:%1.dll** Build a .DLL file. **%1.obj** Include the object file. **db2api.lib** Link with the DB2 library. **-def:%1.def** Module definition file. Refer to your compiler documentation for additional compiler options.

Building embedded SQL stored procedures in C or C++ with configuration files

The configuration file, `stp.icc`, in `sqllib/samples/c` and `sqllib/samples/cpp`, allows you to build DB2 embedded SQL stored procedures in C and C++ on AIX.

To use the configuration file to build the embedded SQL stored procedure shared library `spserver` from the source file `spserver.sqc`, do the following:

1. Set the STP environment variable to the program name by entering:
   - For bash or Korn shell:
     ```
     export STP=spserver
     ```
   - For C shell:
     ```
     setenv STP spserver
     ```
2. If you have an `stp.ics` file in your working directory, produced by building a different program with the `stp.icc` file, delete the `stp.ics` file with this command:
   ```
   rm stp.ics
   ```
   An existing `stp.ics` file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:
   ```
   vacbld stp.icc
   ```

Note: The `vacbld` command is provided by VisualAge C++.

The stored procedure shared library is copied to the server in the path `sqllib/function`.

Next, catalog the stored procedures in the shared library by running the `spcat` script on the server:

```
spcat
```

This script connects to the sample database, uncatalogs the stored procedures 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 call the `spdrop.db2` and `spcreate.db2` scripts individually.

Then, stop and restart the database to allow the new shared library to be recognized. If necessary, set the file mode for the shared library so the DB2 instance can access it.

Once you build the stored procedure shared library, `spserver`, you can build the client application, `spclient`, that calls the stored procedures in it. You can build `spclient` using the configuration file, `emb.icc`.

To call the stored procedures, run the sample client application by entering:
`spclient` *database userid password*

where

database
Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

userid Is a valid user ID.

password
Is a valid password.

The client application accesses the shared library, `spserver`, and executes a number of stored procedure functions on the server database. The output is returned to the client application.

Building user-defined functions in C or C++ with configuration files (AIX)

The configuration file, `udf.icc`, in `sqllib/samples/c` and `sqllib/samples/cpp`, allows you to build user-defined functions in C and C++ on AIX.

To use the configuration file to build the user-defined function program `udfsrv` from the source file `udfsrv.c`, do the following:

1. Set the UDF environment variable to the program name by entering:
   - For bash or Korn shell:
     ```
     export UDF=udfsrv
     ```
   - For C shell:
     ```
     setenv UDF udfsrv
     ```

2. If you have a `udf.ics` file in your working directory, produced by building a different program with the `udf.icc` file, delete the `udf.ics` file with this command:

   ```
   rm udf.ics
   ```

   An existing `udf.ics` file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:

   ```
   vacbld udf.icc
   ```

   Note: The `vacbld` command is provided by VisualAge C++.

The UDF library is copied to the server in the path `sqllib/function`.

If necessary, set the file mode for the user-defined function so the DB2 instance can run it.

Once you build `udfsrv`, you can build the client application, `udfcli`, that calls it. DB2 CLI and embedded SQL versions of this program are provided.

You can build the DB2 CLI `udfcli` program from the source file `udfcli.c`, in `sqllib/samples/cli`, by using the configuration file `cli.icc`.

You can build the embedded SQL `udfcli` program from the source file `udfcli.sqc`, in `sqllib/samples/c`, by using the configuration file `emb.icc`.

To call the UDF, run the sample calling application by entering the executable name:

```
udfcli
```

The calling application calls the `ScalarUDF` function from the `udfsrv` library.

Rebuilding DB2 routine shared libraries

DB2 will cache 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 file" on page 267.
2. **Trusted or threadsafe routines.** Except for SQL routines (including SQL procedures), the only way to ensure that an updated version of a DB2 routine library is picked up when that library is used for trusted, or threadsafe routines, is to recycle the DB2 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 file

This file contains important settings for application development.

The keyword KEEPFENCED has the default value YES. For fenced, not threadsafe routines (stored procedures and UDFs), this keeps the routine process alive. It is best to change the value of this keyword to NO while developing these 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 "Rebuilding DB2 routine shared libraries" on page 266.

Note: KEEPFENCED was known as KEEPDARI in previous versions of DB2.

For Java application development, you need to update the JDK_PATH keyword with the path where the Java Development Kit is installed.

Note: JDK_PATH was known as JDK11_PATH in previous versions of DB2.

To change these settings enter:

```
db2 update dbm cfg using <keyword> <value>
```

For example, to set the keyword KEEPFENCED to NO:

```
db2 update dbm cfg using KEEPFENCED NO
```

To set the JDK_PATH keyword to the directory /home/db2inst/jdk13:

```
db2 update dbm cfg using JDK_PATH /home/db2inst/jdk13
```

To view the current settings in the database manager configuration file, enter:

```
db2 get dbm cfg
```

Note: On Windows, you need to enter these commands in a DB2 command window.

Chapter 7. 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.
```

© Copyright IBM Corp. 1993, 2008

```
           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.

Supported SQL data types in COBOL embedded SQL applications

Certain predefined COBOL data types correspond to DB2 database column types. Only these COBOL data types can be declared as host variables.

The following table shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

Table 38. SQL Data Types Mapped to COBOL Declarations

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
SMALLINT (500 or 501)	01 name PIC S9(4) COMP-5.	16-bit signed integer
INTEGER (496 or 497)	01 name PIC S9(9) COMP-5.	32-bit signed integer
BIGINT (492 or 493)	01 name PIC S9(18) COMP-5.	64-bit signed integer
DECIMAL(*p*,*s*) (484 or 485)	01 name PIC S9(*m*)V9(*n*) COMP-3.	Packed decimal
REAL[2] (480 or 481)	01 name USAGE IS COMP-1.	Single-precision floating point
DOUBLE[3] (480 or 481)	01 name USAGE IS COMP-2.	Double-precision floating point
CHAR(*n*) (452 or 453)	01 name PIC X(*n*).	Fixed-length character string

Table 38. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
VARCHAR(*n*) (448 or 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=32 672	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*n*<=32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (920 or 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	Identifies BLOB entities residing on the server
BLOB file reference variable[4] (916 or 917)	01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.	Descriptor for file containing BLOB data
DATE (384 or 385)	01 identifier PIC X(10).	10-byte character string
TIME (388 or 389)	01 identifier PIC X(8).	8-byte character string
TIMESTAMP (392 or 393)	01 identifier PIC X(26).	26-byte character string
XML[5] (988 or 989)	01 name USAGE IS SQL TYPE IS XML AS CLOB (size).	XML value

The following data types are only available in the DBCS environment.

Table 39. SQL Data Types Mapped to COBOL Declarations

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
GRAPHIC(*n*) (468 or 469)	01 name PIC G(*n*) DISPLAY-1.	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	Variable length double-byte character string with 2-byte string length indicator
LONG VARGRAPHIC (472 or 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 16 337<=*n*<=16 350	Variable length double-byte character string with 2-byte string length indicator
DBCLOB(*n*) (412 or 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n). 1<=*n*<=1 073 741 823	Large object variable-length double-byte character string with 4-byte string length indicator
DBCLOB locator variable[4] (968 or 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (924 or 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Note:

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is a synonym for DOUBLE.
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.
5. The SQL_TYP_XML/SQL_TYP_NXML value is returned by DESCRIBE requests only. It cannot be used directly by the application to bind application resources to XML values.

The following are additional rules for supported COBOL data types:

- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. See the following sample:

```
01 identifier PIC S9(m)V9(n) COMP-3
```

- Use V to denote the decimal point.
- Values for *n* and *m* must be greater than or equal to 1.
- The value for *n* + *m* cannot exceed 31.
- The value for *s* equals the value for *n*.
- The value for *p* equals the value for *n* + *m*.
- The repetition factors *(n)* and *(m)* are optional. The following examples are all valid:

```
01 identifier PIC S9(3)V COMP-3
01 identifier PIC SV9(3) COMP-3
01 identifier PIC S9V COMP-3
01 identifier PIC SV9 COMP-3
```

- PACKED-DECIMAL can be used instead of COMP-3.

- Arrays are *not* supported by the COBOL precompiler.

Building COBOL routines

Compile and link options for COBOL routines

AIX IBM COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the IBM COBOL for AIX compiler on AIX, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile Options:

cob2 The IBM COBOL for AIX compiler.

-qpgmname\(mixed\)
Instructs the compiler to permit CALLs to library entry points with mixed-case names.

-qlib Instructs the compiler to process COPY statements.

-c Perform compile only; no link. Compile and link are separate steps.

-I$DB2PATH/include/cobol_a
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include/cobol_a`.

Link Options:

cob2 Use the compiler to link edit.

-o $1 Specify the output as a shared library file.

$1.o Specify the stored procedure object file.

checkerr.o
Include the utility object file for error-checking.

-bnoentry
Do not specify the default entry point to the shared library.

-bE:$1.exp
Specify an export file. The export file contains a list of the stored procedures.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`.

-ldb2 Link with the database manager library.

Refer to your compiler documentation for additional compiler options.

AIX Micro Focus COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on AIX, as demonstrated in the `bldrtn` build script. Note that the DB2 MicroFocus COBOL include files are found by setting up the COBCPY environment variable, so no -I flag is needed in the compile step. Refer to the bldapp script for an example.

Compile and link options for bldrtn

Compile options:

cob The MicroFocus COBOL compiler.

-c Perform compile only; no link. Compile and link are separate steps.

$EXTRA_COBOL_FLAG="-C MFSYNC"
Enables 64-bit support.

-x Compile to an object module when used with the `-c` option.

Link options:

cob Use the compiler as a front-end for the linker.

-x Produce a shared library.

-o $1 Specify the executable program.

$1.o Specify the program object file.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`.

-ldb2 Link to the DB2 library.

-ldb2gmf
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

HP-UX Micro Focus COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on HP-UX, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

cob The COBOL compiler.

$EXTRA_COBOL_FLAG
Contains "-C MFSYNC" if the HP-UX platform is IA64 and 64-bit support is enabled.

Link options:

-y Specify that the desired output is a shared library.

-o $1 Specify the executable.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries.

-ldb2 Link to the DB2 shared library.

-ldb2gmf
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

Solaris Micro Focus COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on Solaris, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile options:

cob The COBOL compiler.

-cx Compile to object module.

$EXTRA_COBOL_FLAG
For 64-bit support, contains the value "-C MFSYNC"; otherwise it contains no value.

Link options:

cob Use the compiler as a front-end for the linker.

-y Create a self-contained standalone shared library.

-o $1 Specify the executable program.

$1.o Specify the program object file.

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib64`.

-ldb2 Link to the DB2 library.

-ldb2gmf
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

Linux Micro Focus COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on Linux, as demonstrated in the `bldrtn` build script.

Compile and link options for bldrtn

Compile and Link options:

cob The COBOL compiler

$EXTRA_COBOL_FLAG
For 64-bit support, contains the value ″-C MFSYNC″; otherwise it contains no value.

-y Specify to compile to self-contained callable shared object

-o $1 Specify the executable.

$1.cbl Specify the source file

-L$DB2PATH/$LIB
Specify the location of the DB2 runtime shared libraries.

-ldb2 Link to the DB2 library.

-ldb2gmf
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

Windows IBM COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures and user-defined functions) on Windows with the IBM VisualAge COBOL compiler, as demonstrated in the `bldrtn.bat` batch file.

Compile and link options for bldrtn

Compile options:

cob2 The IBM VisualAge COBOL compiler.

-qpgmname(mixed)
Instructs the compiler to permit CALLs to library entry points with mixed-case names.

-c Perform compile only; no link. This batch file has separate compile and link steps.

-qlib Instructs the compiler to process COPY statements.

-I*path* Specify the location of the DB2 include files. For example: `-I"%DB2PATH%\include\cobol_a"`.

%EXTRA_COMPFLAG%
If `"set IBMCOB_PRECOMP=true"` is uncommented, the IBM COBOL precompiler is used to precompile the embedded SQL. It is invoked with one of the following formulations, depending on the input parameters:

-q"SQL('database sample CALL_RESOLUTION DEFERRED')"
precompile using the default sample database, and defer call resolution.

-q"SQL('database %2 CALL_RESOLUTION DEFERRED')"
precompile using a database specified by the user, and defer call resolution.

Link options:

ilink Use the IBM VisualAge COBOL linker.

/free Free format.

/nol No logo.

/dll Create the DLL with the source program name.

db2api.lib
Link with the DB2 library.

%1.exp Include the export file.

%1.obj Include the program object file.

iwzrwin3.obj
Include the object file provided by IBM VisualAge COBOL.

Refer to your compiler documentation for additional compiler options.

Windows Micro Focus COBOL routine compile and link options

The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures and user-defined functions) on Windows with the Micro Focus COBOL compiler, as demonstrated in the `bldrtn.bat` batch file.

Compile and link options for bldrtn

Compile options:

cobol The Micro Focus COBOL compiler.

/case Prevent external symbols being converted to uppercase.

Link options:

cbllink
Use the Micro Focus COBOL linker to link edit.

/d Create a .dll file.

db2api.lib
Link with the DB2 API library.

Refer to your compiler documentation for additional compiler options.

Building IBM COBOL routines on AIX

DB2 provides build scripts for compiling and linking COBOL embedded SQL and DB2 administrative API programs. These are located in the `sqllib/samples/cobol` directory, along with sample programs that can be built with these files.

The script, `bldrtn`, in `sqllib/samples/cobol`, contains the commands to build routines (stored procedures). The script compiles the routines into a shared library that can be called by a client application.

The first parameter, `$1`, specifies the name of your source file. The second parameter, `$2`, specifies the name of the database to which you want to connect. Since the shared library must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. The script uses the source file name, `$1`, for the shared library name. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

To build the sample program `outsrv` from the source file `outsrv.sqb`, connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also include the database name:

```
bldrtn outsrv database
```

The script file copies the shared library to the server in the path `sqllib/function`.

Once you build the routine shared library, `outsrv`, you can build the client application, `outcli`, that calls the routine within the library. You can build `outcli` using the script file `bldapp`.

To call the routine, run the sample client application by entering:

```
outcli database userid password
```

where

database
Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the shared library, outsrv, and executes the routine of the same name on the server database, and then returns the output to the client application.

Building UNIX Micro Focus COBOL routines

DB2 provides build scripts for compiling and linking Micro Focus COBOL embedded SQL and DB2 API programs. These are located in the sqllib/samples/cobol_mf directory, along with sample programs that can be built with these files.

The script, bldrtn, contains the commands to build routines (stored procedures). The script compiles the routine source file into a shared library that can be called by a client application.

The first parameter, $1, specifies the name of your source file. The script uses the source file name for the shared library name. The second parameter, $2, specifies the name of the database to which you want to connect. Since the shared library must be built in the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default sample database.

Before building Micro Focus COBOL routines, you must run the following commands:

```
db2stop
db2set DB2LIBPATH=$LD_LIBRARY_PATH
db2set DB2ENVLIST="COBDIR LD_LIBRARY_PATH"
db2set
db2start
```

Ensure that db2stop stops the database. The last db2set command is issued to check your settings: make sure DB2LIBPATH and DB2ENVLIST are set correctly.

To build the sample program outsrv from the source file outsrv.sqb, if connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also enter the database name:

```
bldrtn outsrv database
```

The script file copies the shared library to the server in the path sqllib/function.

Once you build the stored procedure outsrv, you can build the client application outcli that calls it. You can build outcli using the script file, bldapp.

To call the stored procedure, run the sample client application by entering:

```
outcli database userid password
```

where

database
Is the name of the database to which you want to connect. The name could be sample, or its alias, or another name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the shared library, `outsrv`, and executes the stored procedure function of the same name on the server database. The output is then returned to the client application.

Building IBM COBOL routines on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in IBM COBOL. These are located in the `sqllib\samples\cobol` directory, along with sample programs that can be built with these files.

DB2 supports two precompilers for building IBM COBOL applications on Windows, the DB2 precompiler and the IBM COBOL precompiler. The default is the DB2 precompiler. The IBM COBOL precompiler can be selected by uncommenting the appropriate line in the batch file you are using. Precompilation with IBM COBOL is done by the compiler itself, using specific precompile options.

The batch file, `bldrtn.bat`, contains the commands to build embedded SQL routines (stored procedures). The batch file compiles the routines into a DLL on the server. It takes two parameters, represented inside the batch file by the variables `%1` and `%2`.

The first parameter, `%1`, specifies the name of your source file. The batch file uses the source file name, `%1`, for the DLL name. The second parameter, `%2`, specifies the name of the database to which you want to connect. Since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

If using the default DB2 precompiler, `bldrtn.bat` passes the parameters to the precompile and bind file, `embprep.bat`.

If using the IBM COBOL precompiler, `bldrtn.bat` copies the `.sqb` source file to a `.cbl` source file. The compiler performs the precompile on the `.cbl` source file with specific precompile options.

To build the sample program `outsrv` from the source file `outsrv.sqb`, connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also include the database name:

```
bldrtn outsrv database
```

The batch file copies the DLL to the server in the path `sqllib\function`.

Once you build the DLL `outsrv`, you can build the client application `outcli` that calls the routine within the DLL (which has the same name as the DLL). You can build `outcli` using the batch file `bldapp.bat`.

To call the `outsrv` routine, run the sample client application by entering:

```
outcli database userid password
```

where

database
Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the DLL, `outsrv`, and executes the routine of the same name on the server database, and then returns the output to the client application.

Building Micro Focus COBOL routines on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in Micro Focus COBOL. These are located in the `sqllib\samples\cobol_mf` directory, along with sample programs that can be built with these files.

The batch file `bldrtn.bat` contains the commands to build embedded SQL routines (stored procedures). The batch file compiles the routines into a DLL on the server. The batch file takes two parameters, represented inside the batch file by the variables `%1` and `%2`.

The first parameter, `%1`, specifies the name of your source file. The batch file uses the source file name, `%1`, for the DLL name. The second parameter, `%2`, specifies the name of the database to which you want to connect. Since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

To build the sample program `outsrv` from the source file `outsrv.sqb`, if connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also enter the database name:

```
bldrtn outsrv database
```

The script file copies the DLL to the server in the path `sqllib/function`.

Once you build the DLL, `outsrv`, you can build the client application, `outcli`, that calls the routine within the DLL (which has the same name as the DLL). You can build `outcli` using the batch file, `bldapp.bat`.

To call the `outsrv` routine, run the sample client application by entering:

```
outcli database userid password
```

where

database
Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another name.

userid Is a valid user ID.

password
Is a valid password for the user ID.

The client application accesses the DLL, `outsrv`, which executes the routine of the same name on the server database. The output is then returned to the client application.

Chapter 8. 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:

JDBC An application programming interface for accessing data in databases that supports dynamic SQL execution.

SQLJ An application programming interface for accessing data in databases that supports dynamic and static SQL execution.

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.

Before developing a Java routine, it is important to both understand the basics of routines and the unique features of Java routines. An understanding of Java database application development using either JDBC or SQLJ is also important. To learn more about these subjects, see the topics:

- "External routine creation" on page 112
- "Java application development for DB2" in *Developing Java Applications*

Supported Java routine development software

To develop and deploy external routines in Java you must use supported Java development software.

© IBM Corporation 1993, 2008

The minimum supported software development kit (SDK) version for Java routine development is:

- IBM SDK for Java 1.4.2

The maximum supported SDK version for Java routine development is:

- IBM SDK for Java 5

It is recommended that you use the SDK for Java that is installed with the DB2 Database for Linux, UNIX, and Windows product, however an alternate SDK for Java can be specified, however the SDK must be of the same bit-width as the DB2 instance.

All other development software supported for DB2 database application development in Java can be used for external routine development in Java.

JDBC and SQLJ application programming interface support for Java routines

External routines developed in Java can make use of the following application programming interfaces (APIs):

- JDBC
- SQLJ

Starting in DB2 Version 9.5, the default driver is the IBM Data Server Driver for JDBC and SQLJ. This supports both the JDBC and SQLJ APIs and can be used to develop external Java routines.

The IBM Data Server Type 2 Driver can also be used for Java routine development but only provides support for JDBC Type 2. It is strongly recommended that you use the IBM Data Server Driver for JDBC and SQLJ as the IBM Data Server Type 2 Driver is deprecated.

The procedures for implementing Java routines are the same regardless which API or driver is used.

Specification of an SDK for Java routine development (UNIX)

To build and run Java routine code in UNIX environments, the DB2 database manager configuration parameter JDK_PATH must be set to the path of an SDK for Java that is installed on the DB2 database server. If the installation process for the DB2 Database for Linux, UNIX, and Windows product installs an SDK for Java, by default the JDK_PATH parameter is set to the path of that SDK for Java:

- `$INSTDIR/sqllib/java/jdk32` for DB2 database servers installed in Linux on x86 environments
- `$INSTDIR/sqllib/java/jdk64` for DB2 database servers that provide an SDK installed in all other environments

The JDK_PATH parameter value can be changed to specify another SDK for Java installed on the computer, however the SDK must be of the same bit-width as the DB2 instance.

Prerequisites

- User must have access to the DB2 database server.

- User must have the authority to read and update the database manager configuration file.
- User must have authority to install an SDK for Java in the file system where the DB2 instance is installed.

Procedure

1. Check the JDK_PATH parameter value by issuing the following command from a 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 SDK for Java, install it on the DB2 database server and note the installation path. You will require the install path value for the next step.
3. Update the JDK_PATH parameter value by issuing the following command from a DB2 Command Window where <path> is the path where the other SDK for Java is installed:

   ```
   db2 update dbm cfg using JDK_PATH <path>
   ```
4. Stop and restart your DB2 instance by issuing the following DB2 commands from a DB2 Command Window:

   ```
   db2stop;

   db2start;
   ```
5. Verify that the JDK_PATH parameter value is correctly set by issuing the following command from a DB2 Command Window:

   ```
   db2 get dbm cfg
   ```

After you complete these steps the specified SDK for Java will be used for building and running Java routines. The CLASSPATH, PATH, and LIBPATH environment variables within the environment are set automatically.

Specification of a driver for Java routines

Java routine development and invocation requires that a JDBC or SQLJ driver be specified. Java routines can use one of the following two drivers:

- IBM Data Server Driver for JDBC and SQLJ
- DB2 Type 2 Driver

By default, DB2 uses the IBM Data Server Driver for JDBC and SQLJ. This driver is preferred because it is more robust, the DB2 Type 2 Driver is deprecated, and because it is a prerequisite if Java routines contain:

- Parameters of data type XML
- Variables of data type XML
- References to XML data
- References to XML functions
- Any other native-XML feature

If problems occur in the migration of existing Java routines, the DB2_USE_DB2JCCT2_JROUTINE DB2 environment variable can be set to the value, NO, to use the legacy IBM DB2 Type 2 Driver. This can be done by issuing the following command from a DB2 command window:

```
db2set DB2_USE_DB2JCCT2_JROUTINE=NO
```

After issuing this command, you must stop and restart the DB2 instance in order for the change to take effect.

Tools for developing Java (JDBC and SQLJ) routines

Tools make Java routine development go quickly and easily. The following DB2 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:

- DB2 Command Line Processor
- DB2 Command Window

Other IBM software products provide graphical tools for developing Java routines including:

- 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.

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 21

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*

With the prerequisite knowledge, designing Java routines consists mainly of learning about the unique features and characteristics of Java routines:

- "Supported SQL data types in Java routines" on page 289
- "Parameters in Java routines" on page 291
- "Parameter style JAVA procedures" on page 292
- "Parameter style JAVA Java functions and methods" on page 293
- "Returning result sets from JDBC procedures" on page 303
- "Returning result sets from SQLJ procedures" on page 303
- "Restrictions on Java routines" on page 306
- "Table function execution model for Java" on page 135

After having learned about the Java characteristics, you might want to refer to:

- "Creating Java routines from the command line" on page 309

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. The DB2 Driver for JDBC and SQLJ converts the data exchanged between Java source code and a DB2 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 40. SQL Data Types Mapped to Java Declarations

SQL Column Type	Java Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short, boolean	16-bit, signed integer
INTEGER (496 or 497)	int	32-bit, signed integer
BIGINT [1] (492 or 493)	long	64-bit, signed integer
REAL (480 or 481)	float	Single precision floating point
DOUBLE (480 or 481)	double	Double precision floating point
DECIMAL(*p*,*s*) (484 or 485)	java.math.BigDecimal	Packed decimal
CHAR(*n*) (452 or 453)	java.lang.String	Fixed-length character string of length *n* where *n* is from 1 to 254
CHAR(*n*) FOR BIT DATA	byte[]	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	java.lang.String	Variable-length character string
VARCHAR(*n*) FOR BIT DATA	byte[]	Variable-length character string
LONG VARCHAR (456 or 457)	java.lang.String	Long variable-length character string

Table 40. SQL Data Types Mapped to Java Declarations (continued)

SQL Column Type	Java Data Type	SQL Column Type Description
LONG VARCHAR FOR BIT DATA	byte[]	Long variable-length character string
BLOB(*n*) (404 or 405)	java.sql.Blob	Large object variable-length binary string
CLOB(*n*) (408 or 409)	java.sql.Clob	Large object variable-length character string
DBCLOB(*n*) (412 or 413)	java.sql.Clob	Large object variable-length double-byte character string
DATE (384 or 385)	java.sql.Date	10-byte character string
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	26-byte character string
GRAPHIC(*n*) (468 or 469)	java.lang.String	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	java.lang.String	Non-null-terminated varying double-byte character string with 2-byte string length indicator
LONGVARGRAPHIC (472 or 473)	java.lang.String	Non-null-terminated varying double-byte character string with 2-byte string length indicator
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.

Note:

1. For Java applications connected from a DB2 Universal Database Version 8.1 client to a DB2 Universal Database Version 7.1 (or 7.2) server, note the following: when the getObject() method is used to retrieve a BIGINT value, a java.math.BigDecimal object is returned.
2. Parameters of an SQL array data type are mapped to class com.ibm.db2.ARRAY.
3. LONG VARCHAR, LONG VARGRAPHIC, XML, REFERENCE, UDT and ARRAY are not supported for the ARRAY data type.

Connection contexts in SQLJ routines

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

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

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

The SQLJ routine above must be changed as follows:

```
#context MyContext;

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

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

Parameters in Java routines

Parameter declaration in Java routines must conform to the requirements of one of the supported parameter styles. The following two 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 will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification.

There are some Java routine features that cannot be implemented or used with PARAMETER STYLE JAVA. These are as follows:

- 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

If you need to implement the above features you can implement your routine in C, or write it in Java, using parameter style DB2GENERAL.

Aside from these specific cases, you should always create and implement your Java routines using PARAMETER STYLE JAVA.

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` system-defined 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. Once you have created and registered them, you can call them from programs in any language. Typically, you can call JDBC APIs from your stored procedures, but you cannot call them from UDFs.

When developing routines in Java, it is strongly recommended that you register them using the PARAMETER STYLE JAVA clause in the CREATE statement. PARAMETER STYLE DB2GENERAL is still available to enable the implementation of the following features in Java routines:

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

If you have PARAMETER STYLE DB2GENERAL routines that do not use any of the above features, it is recommended that you migrate them to PARAMETER STYLE JAVA for portability.

DB2GENERAL UDFs

You can create and use UDFs in Java just as you would in other languages, with only a few minor differences when compared to C UDFs. After you code the UDF, you register it with the database. You can then refer to it in your applications.

In general, if you declare a UDF taking arguments of SQL types *t1*, *t2*, and *t3*, returning type *t4*, it will be called as a Java method with the expected Java signature:

```
public void name ( T1 a, T2 b, T3 c, T4 d)   {.....}
```

Where:

- *name* is the Java method name
- *T1* through *T4* are the Java types that correspond to SQL types *t1* through *t4*.
- *a*, *b*, and *c* are variable names for the input arguments.
- *d* is an variable name that represents the output argument.

For example, given a UDF called `sample!test3` that returns INTEGER and takes arguments of type CHAR(5), BLOB(10K), and DATE, DB2 expects the Java implementation of the UDF to have the following signature:

```
import COM.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3,
                     int result)  {... }
}
```

Java routines that implement table functions require more arguments. Beside the variables representing the input, an additional variable appears for each column in the resulting row. For example, a table function can be declared as:

```
public void test4(String arg1, int result1,
                  Blob result2, String result3);
```

SQL NULL values are represented by Java variables that are not initialized. These variables have a value of zero if they are primitive types, and Java null if they are object types, in accordance with Java rules. To tell an SQL NULL apart from an ordinary zero, you can call the function `isNull` for any input argument:

```
{ ....
   if (isNull(1)) { /* argument #1 was a SQL NULL */ }
   else           { /* not NULL */ }
}
```

In the above example, the argument numbers start at one. The `isNull()` function, like the other functions that follow, are inherited from the `COM.ibm.db2.app.UDF` class.

To return a result from a scalar or table UDF, use the `set()` method in the UDF, as follows:

```
{ ....
   set(2, value);
}
```

Where '2' is the index of an output argument, and value is a literal or variable of a compatible type. The argument number is the index in the argument list of the selected output. In the first example in this section, the `int result` variable has an index of 4; in the second, `result1` through `result3` have indices of 2 through 4.

Like C modules used in UDFs and stored procedures, you cannot use the Java standard I/O streams (`System.in`, `System.out`, and `System.err`) in Java routines.

Remember that all the Java class files (or the JARs that contain the classes) that you use to implement a routine must reside in the `sqllib/function` directory, or in a directory specified in the database manager's CLASSPATH.

Typically, DB2 calls a UDF many times, once for each row of an input or result set in a query. If SCRATCHPAD is specified in the CREATE FUNCTION statement of the UDF, DB2 recognizes that some "continuity" is needed between successive invocations of the UDF, and therefore the implementing Java class is not instantiated for each call, but generally speaking once per UDF reference per statement. Generally it is instantiated before the first call and used thereafter, but can for table functions be instantiated more often. If, however, NO SCRATCHPAD is specified for a UDF, either a scalar or table function, then a clean instance is instantiated for each call to the UDF.

A scratchpad can be useful for saving information across calls to a UDF. While Java and OLE UDFs can either use instance variables or set the scratchpad to

achieve continuity between calls, C and C++ UDFs must use the scratchpad. Java UDFs access the scratchpad with the getScratchPad() and setScratchPad() methods available in `COM.ibm.db2.app.UDF`.

For Java table functions that use a scratchpad, control when you get a new scratchpad instance by using the FINAL CALL or NO FINAL CALL option on the CREATE FUNCTION statement.

The ability to achieve continuity between calls to a UDF by means of a scratchpad is controlled by the SCRATCHPAD and NO SCRATCHPAD option of CREATE FUNCTION, regardless of whether the DB2 scratchpad or instance variables are used.

For scalar functions, you use the same instance for the entire statement.

Note that every reference to a Java UDF in a query is treated independently, even if the same UDF is referenced multiple times. This is the same as what happens for OLE, C and C++ UDFs as well. At the end of a query, if you specify the FINAL CALL option for a scalar function then the object's close() method is called. For table functions the close() method will always be invoked as indicated in the subsection which follows this one. If you do not define a close() method for your UDF class, then a stub function takes over and the event is ignored.

If you specify the ALLOW PARALLEL clause for a Java UDF in the CREATE FUNCTION statement, DB2 may elect to evaluate the UDF in parallel. If this occurs, several distinct Java objects may be created on different partitions. Each object receives a subset of the rows.

As with other UDFs, Java UDFs can be FENCED or NOT FENCED. NOT FENCED UDFs run inside the address space of the database engine; FENCED UDFs run in a separate process. Although Java UDFs cannot inadvertently corrupt the address space of their embedding process, they can terminate or slow down the process. Therefore, when you debug UDFs written in Java, you should run them as FENCED UDFs.

Supported SQL data types in DB2GENERAL routines

When you call PARAMETER STYLE DB2GENERAL routines, DB2 converts SQL types to and from Java types for you. Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 41. DB2 SQL Types and Java Objects

SQL Column Type	Java Data Type
SMALLINT	short
INTEGER	int
BIGINT	long
REAL[1]	float
DOUBLE	double
DECIMAL(p,s)	java.math.BigDecimal
NUMERIC(p,s)	java.math.BigDecimal
CHAR(*n*)	java.lang.String
CHAR(*n*) FOR BIT DATA	COM.ibm.db2.app.Blob

Table 41. DB2 SQL Types and Java Objects (continued)

SQL Column Type	Java Data Type
VARCHAR(*n*)	java.lang.String
VARCHAR(*n*) FOR BIT DATA	COM.ibm.db2.app.Blob
LONG VARCHAR	java.lang.String
LONG VARCHAR FOR BIT DATA	COM.ibm.db2.app.Blob
GRAPHIC(*n*)	java.lang.String
VARGRAPHIC(*n*)	String
LONG VARGRAPHIC[2]	String
BLOB(*n*)[2]	COM.ibm.db2.app.Blob
CLOB(*n*)[2]	COM.ibm.db2.app.Clob
DBCLOB(*n*)[2]	COM.ibm.db2.app.Clob
DATE[3]	String
TIME[3]	String
TIMESTAMP[3]	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

This interface provides the following routine to fetch a JDBC connection to the embedding application context:

```
public java.sql.Connection getConnection()
```

You can use this handle to run SQL statements. Other methods of the `StoredProc` interface are listed in the file `sqllib/samples/java/StoredProc.java`.

There are five classes/interfaces that you can use with Java Stored Procedures or UDFs:

- COM.ibm.db2.app.StoredProc

- COM.ibm.db2.app.UDF
- COM.ibm.db2.app.Lob
- COM.ibm.db2.app.Blob
- COM.ibm.db2.app.Clob

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 PARAMETER STYLE DB2GENERAL UDFs must be public and must implement this Java interface. You must declare such a class as follows:

```
public class user-UDF-class extends COM.ibm.db2.app.UDF{ ... }
```

You can only call methods of the `COM.ibm.db2.app.UDF` interface in the context of the currently executing UDF. For example, you cannot use operations on LOB arguments, result- or status-setting calls, etc., after a UDF returns. A Java exception will be thrown if this rule is violated.

Argument-related calls use a column index to identify the column being set. These start at 1 for the first argument. Output arguments are numbered higher than the input arguments. For example, a scalar UDF with three inputs uses index 4 for the output.

Any exception returned from the UDF is caught by the database and returned to the caller with SQLCODE -4302, SQLSTATE 38501.

The following methods are associated with the `COM.ibm.db2.app.UDF` class:

```
public UDF() [default constructor]
```

This constructor is called by the database at the beginning of a series of UDF calls. It precedes the first call to the UDF.

```
public void close()
```

This function is called by the database at the end of a UDF evaluation, if the UDF was created with the FINAL CALL option. It is analogous to the final call for a C UDF. For table functions, close() is called after the CLOSE call to the UDF method (if NO FINAL CALL is coded or defaulted), or after the FINAL call (if FINAL CALL is coded). If a Java UDF class does not implement this function, a no-op stub will handle and ignore this event.

```
public int getCallType() throws Exception
```

Table function UDF methods use getCallType() to find out the call type for a particular call. It returns a value as follows (symbolic defines are provided for these values in the COM.ibm.db2.app.UDF class definition):

- -2 FIRST call
- -1 OPEN call
- 0 FETCH call
- 1 CLOSE call
- 2 FINAL call

```
public boolean isNull(int) throws Exception
```

This function tests whether an input argument with the given index is an SQL NULL.

```
public boolean needToSet(int) throws Exception
```

This function tests whether an output argument with the given index needs to be set. This can be false for a table UDF declared with DBINFO, if that column is not used by the UDF caller.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

This function sets the output argument with the given index to the given value. The index has to refer to a valid output argument, the data type must match, and the value must have an acceptable length and contents. Strings with Unicode characters must be representable in the database code page. Errors result in an exception being thrown.

```
public void setSQLstate(String) throws Exception
```

This function can be called from a UDF to set the SQLSTATE to be returned from this call. A table UDF should call this function with "02000" to signal the end-of-table condition. If the string is not acceptable as an SQLSTATE, an exception will be thrown.

```
public void setSQLmessage(String) throws Exception
```

This function is similar to the `setSQLstate` function. It sets the SQL message result. If the string is not acceptable (for example, longer than 70 characters), an exception will be thrown.

```
public String getFunctionName() throws Exception
```

This function returns the name of the executing UDF.

```
public String getSpecificName() throws Exception
```

This function returns the specific name of the executing UDF.

```
public byte[] getDBinfo() throws Exception
```

This function returns a raw, unprocessed DBINFO structure for the executing UDF, as a byte array. You must first declare it with the DBINFO option.

```
public String getDBname() throws Exception
public String getDBauthid() throws Exception
public String getDBtbschema() throws Exception
public String getDBtbname() throws Exception
public String getDBcolname() throws Exception
public String getDBver_rel() throws Exception
public String getDBplatform() throws Exception
public String getDBapplid() throws Exception
```

These functions return the value of the appropriate field from the DBINFO structure of the executing UDF.

```
public int getDBprocid() throws Exception
```

This function returns the routine id of the procedure which directly or indirectly invoked this routine. The routine id matches the ROUTINEID column in SYSCAT.ROUTINES which can be used to retrieve the name of the invoking procedure. If the executing routine is invoked from an application, getDBprocid() returns 0.

```
public int[] getDBcodepg() throws Exception
```

This function returns the SBCS, DBCS, and composite code page numbers for the database, from the DBINFO structure. The returned integer array has the respective numbers as its first three elements.

```
public byte[] getScratchpad() throws Exception
```

This function returns a copy of the scratchpad of the currently executing UDF. You must first declare the UDF with the SCRATCHPAD option.

```
public void setScratchpad(byte[]) throws Exception
```

This function overwrites the scratchpad of the currently executing UDF with the contents of the given byte array. You must first declare the UDF with the SCRATCHPAD option. The byte array must have the same size as `getScratchpad()` returns.

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 representede in this code page, and can cause an exception to be thrown during conversion. This can happen during an append operation, or during a UDF or StoredProc `set()` call. This is necessary to hide the distinction between a CLOB and a DBCLOB from the Java programmer.

The following methods are associated with the `COM.ibm.db2.app.Clob` class:

```
public long size() throws Exception
```

This function returns the length (in characters) of the CLOB.

```
public java.io.Reader getReader() throws Exception
```

This function returns a new Reader to read the contents of the CLOB or DBCLOB. Efficient seek/mark operations are available on that object.

```
public java.io.Writer getWriter() throws Exception
```

This function returns a new Writer to append characters to this CLOB or DBCLOB. Appended characters become immediately visible on all existing Reader instances produced by this object's `GetReader()` call.

Passing parameters of data type ARRAY to Java routines

The ability to pass parameters of data type ARRAY to and from Java procedures is supported in DB2 9.5.

You would choose to implement the use of arrays being passed between applications and Java stored procedures if you want to pass:

- large amounts of homogenous data to a procedure using a single parameter.
- a variable number of inputs of the same data type to a procedure using only 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 using a single parameter. If you did not use the ARRAY data type, to do this, you would require one parameter per student name.

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. For example, if an ARRAY data type named `IntArray` is created by executing the CREATE TYPE statement, to pass a parameter of this type to a procedure, do the following:

```
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 example below. 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];
}
}
```

For additional information, see the *Related links* section.

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.

To return a result set from a JDBC procedure:

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

   ```
   public static void getHighSalaries(
     double inSalaryThreshold,         // double input
     int[] errorCode,                  // SQLCODE output
     ResultSet[] rs)                   // ResultSet output
   ```

2. Open the invoker's database connection (using a Connection object):

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

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

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

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

4. Execute the statement:

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

5. End the procedure body.

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

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.

To return a result set from an SQLJ procedure:

1. Declare an iterator class to handle query data. For example:

   ```
   #sql iterator SpServerEmployees(String, String, double);
   ```

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

```
public static void getHighSalaries(
  double inSalaryThreshold,         // double input
  int[] errorCode,                  // SQLCODE output
  ResultSet[] rs)                   // ResultSet output
```

3. Instantiate an iterator object. For example:

```
SpServerEmployees c1;
```

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

```
#sql c1 = {SELECT name, job, CAST(salary AS DOUBLE)
             FROM staff
             WHERE salary > :inSalaryThreshold
             ORDER BY salary};
```

5. Execute the statement and get the result set:

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

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

Receiving procedure result sets in JDBC applications and routines

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

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

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

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

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

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

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

3. Call the procedure:

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

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

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

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

```
      System.out.print(rs.getString(i));
      if (i != numOfColumns)
      {
        System.out.print(", ");
      }
    }
    System.out.println();
  }
```

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

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

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

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

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

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

Receiving procedure result sets in SQLJ applications and routines

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

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

The following restrictions apply to Java routines:

- The system-defined 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 using the DB2 Universal JDBC Driver. This driver does not support this procedure.

  The recommended alternative is to use the DB2 Command 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 with parameter style JAVA:
  - table functions
  - scratchpads in functions
  - access to the DBINFO structure in functions
  - FINAL CALL invocation in functions

  The recommended alternative if you need these features is to create a Java function using parameter style DB2GENERAL or to create the function using 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 defined as NOT FENCED will be invoked as if it had been defined 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.

Table function execution model for Java

For table functions written in Java and using PARAMETER STYLE DB2GENERAL, it is important to understand what happens at each point in DB2's processing of a given statement. The following table details this information for a typical table function. Covered are both the NO FINAL CALL and the FINAL CALL cases, assuming SCRATCHPAD in both cases.

Point in scan time	**NO FINAL CALL LANGUAGE JAVA SCRATCHPAD**	**FINAL CALL LANGUAGE JAVA SCRATCHPAD**
Before the first OPEN for the table function	No calls.	• Class constructor is called (means new scratchpad). UDF method is called with FIRST call. • Constructor initializes class and scratchpad variables. Method connects to Web server.
At each OPEN of the table function	• Class constructor is called (means new scratchpad). UDF method is called with OPEN call. • Constructor initializes class and scratchpad variables. Method connect to Web server, and opens the scan for Web data.	• UDF method is opened with OPEN call. • Method opens the scan for whatever Web data it wants. (Might be able to avoid reopen after a CLOSE reposition, depending on what is saved in the scratchpad.)
At each FETCH for a new row of table function data	• UDF method is called with FETCH call. • Method fetches and returns next row of data, or EOT.	• UDF method is called with FETCH call. • Method fetches and returns new row of data, or EOT.
At each CLOSE of the table function	• UDF method is called with CLOSE call. `close()` method if it exists for class. • Method closes its Web scan and disconnects from the Web server. `close()` does not need to do anything.	• UDF method is called with CLOSE call. • Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist.
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.
2. For table functions with NO SCRATCHPAD specified, the calls to the UDF method are as indicated in this table, but because the user is not asking for any continuity with a scratchpad, DB2 will cause a new object to be instantiated before each call, by calling the class constructor. It is not clear that table functions with NO SCRATCHPAD (and thus no continuity) can do useful things, but they are supported.

Creating Java routines

Creating Java routines consists of:

- Executing a CREATE statement that defines the routine in a DB2 database server
- Developing the routine implementation that corresponds to the routine definition

The ways in which you can create Java routines follow:

- Using the IBM Data Studio
- Using the DB2 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.

Prerequisites

- Review Chapter 8, “Java routines,” on page 285.
- Ensure that you have access to a DB2 Version 9 server, including instances and databases.
- Ensure that the operating system is at a version level that is supported by DB2 database products.
- Ensure that the “Supported Java routine development software” on page 285 is at a version level that is supported for Java routine development.
- Ensure a valid “Specification of a driver for Java routines” on page 287 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 306

Create Java routines using one of the following procedures:

- Creating Java routines using IBM Data Studio
- Creating Java routines using Rational Application Developer
- “Creating Java routines from the command line” on page 309

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. This task comprises a few steps including the formulation of the CREATE statement for the routine, the coding and compilation (translation) of the routine implementation, and the deployment of the Java class to a DB2 database server.

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.

Prerequisites

- Review Chapter 8, "Java routines," on page 285.
- Ensure that you have access to a DB2 Version 9 database server, including instances and databases.
- Ensure that the operating system is at a version level that is supported by DB2 database products.
- Ensure that the "Supported Java routine development software" on page 285 is at a version level that is supported for Java routine development.
- Ensure a valid "Specification of a driver for Java routines" on page 287 development.
- Authority to execute the CREATE PROCEDURE or CREATE FUNCTION statement.

Procedure

1. Code the routine logic in Java.
   - For general information about Java routines and Java routine features, see the topics referenced in the Prerequisites section.
   - 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 291
     - "Parameter style JAVA procedures" on page 292
     - "Parameter style JAVA Java functions and methods" on page 293
   - Declare variables in the same manner as is done for Java database applications. Be careful to correctly use data types that map to DB2 SQL data types. For more on data type mapping between DB2 and Java data types see:
     - "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 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 303
  - "Returning result sets from SQLJ procedures" on page 303
- 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 DB2 database server or install the JAR file to the DB2 database server. For information on how to do this, see:
   - "Deploying Java routine class files to DB2 database servers" on page 314
   - "JAR file administration on the database server" on page 315

   It is recommended that you store class files associated with DB2 routines in the *function directory*. To find out more about the function directory, see information related to the 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 DB2 will look 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.

To invoke your Java routine, see Chapter 11, "Routine invocation," on page 333

Building Java routine code

Once Java routine implementation code has been written, it must be built before the routine assembly can be deployed and the routine invoked. The steps required to build Java routines are similar to those required to build any external routine however there are some differences.

There are a few ways to build Java routines:

- Using the graphical tools provided with the IBM Data Studio
- Using the graphical tools provided within IBM Rational Application Developer
- Using DB2 sample build scripts
- Entering commands from a DB2 Command Window

The graphical tools and DB2 can be customized to build Java routines for a variety of operating systems and with a variety of settings. The sample build scripts and batch files for routines are designed for building DB2 sample routines (procedures and user-defined functions) for a particular operating system using the default supported development software.

There is a separate set of DB2 sample build scripts and batch files for Java routines created with JDBC and SQLJ. In general it is easiest to build Java routines using the graphical tools or the build scripts which can easily be modified if required, however it is often helpful to know how to build routines from DB2 Command Window as well.

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.

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
- To build and run the SpServer.java stored procedure on the server, from the command line:
  1. Compile SpServer.java to produce the file SpServer.class with this command:
     ```
     javac SpServer.java
     ```
  2. Copy `SpServer.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.
  3. 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.
  4. 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.
  5. 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:

1. Compile `UDFsrv.java` to produce the file UDFsrv.class with this command:
   ```
   javac UDFsrv.java
   ```
2. Copy `UDFsrv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.
3. 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:
  1. Compile `UDFsqlsv.java` to produce the file `UDFsqlsv.class` with this command:
     ```
     javac UDFsqlsv.java
     ```
  2. Copy `UDFsqlsv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.
  3. 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 SQL 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.

The following steps demonstrate how to build and run the `SpServer` sample SQLJ stored procedure. These steps use the build file, `bldsqljs` (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 UNIX, and %1, %2, %3, %4, %5, and %6 on Windows. 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.

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` (UNIX) or `bldsqlj.bat` (Windows).

Compile and link options for Java (SQLJ) routines

SQLJ routine options for UNIX

The `bldsqljs` build script builds SQLJ routines on 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 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.

Recommendation: Use the same SQLJ translator and customizer options that `bldsqljs.bat` uses 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. These are the options DB2 recommends that you use 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 DB2 for Java 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 DB2 database servers

Java routine implementations must be deployed to the DB2 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 containing 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 you have installed in the DB2 database.

Prerequisites:

- Identify the DB2 data server to which you want to deploy the routine class.
- On UNIX operating systems, identify the user ID of the DB2 instance owner. If this is not known, see your database administrator.

To deploy individual Java routine class files:

-

  Copy the Java routine class into the DB2 function directory.

  On UNIX operating systems, the function directory is defined as: <install_path>/function where <install_path> is the DB2 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 containing Java routine class files:

-

  You must install the JAR file containing Java routine class files to the DB2 database server file system.

Once the Java routine class files have been 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 DB2 database servers with dependent classes

When the Java routine class files have dependencies on classes that are not part of the standard Java or DB2 classes, repeat the steps identified in the previous section for each dependent class.

Alternatively, the DB2 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 server will automatically search the specified directories in the CLASSPATH system environment variable. On UNIX operating systems, the database server will search 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 DB2 database server. This can be done from an IBM Data Server Client by using system-defined routines that install, replace, or remove JAR files on the DB2 database server.

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

Install

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

Note: The privileges held by the authorization ID of the caller of sqlj.install_jar must include at least one of the following:

- CREATEIN privilege for the implicitly or explicitly specified schema
- SYSADM or DBADM authority

Replace

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

Remove

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

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

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

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

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

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

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

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

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

To remove MYJAR from the database catalogs:

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

Note: On Windows operating systems, DB2 stores JAR files in the path specified by the *DB2INSTPROF* instance-specific registry setting. To make JAR files unique for an instance, you must specify a unique value for *DB2INSTPROF* for that instance.

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 DB2 database server. To ensure that DB2 database manager uses the new version of the Java routine, you must execute a system-defined procedure that will load the new version of the Java class into memory.

To update Java routine classes:

1. Deploy the new Java class or JAR file to the DB2 database server.
2. Execute the following system-defined procedure for fenced routines:

   ```
   CALL SQLJ.REFRESH_CLASSES()
   ```

   This will force the DB2 database manager to load the new class into memory upon the next commit or rollback operation.

   For unfenced routines, this step will not work. For unfenced routines, you must explicitly stop and re-start the DB2 database manager in order for new versions of Java routine classes to be loaded and used.

If you do not perform the steps above, after you update Java routine classes, the DB2 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. The following topics contain examples of Java procedures and functions:

- 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 example below 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 start creating and using 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 working with this Java procedure example you might want to read the following topics:

- Java routines
- Routines
- Building Java routine code

The examples below makes use of a table named `xmlDataTable` defined and containing data 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 example as references when making your own Java procedures:

- Table 42 on page 319
- Table 43 on page 319

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:

Table 42. Java external code file 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 will be used.

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

Example 1: Parameter style JAVA procedure with XML parameters

This example shows the following:

- CREATE PROCEDURE statement for a parameter style JAVA procedure
- Java code for a parameter style JAVA procedure with XML parameters

This procedure takes an input parameter, `inXML`, inserts a row including that value into a table, queriesXML data using both an SQL statement and an XQuery expression, and sets two output parameters, `outXML1`, and `outXML2`.

Table 43. Code to create a parameter style JAVA procedure with XML parameters

```
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'@
```

Table 43. Code to create a parameter style JAVA procedure with XML parameters (continued)

```
//*************************************************************************
//  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 ;
}
```

Chapter 9. OLE automation routine design

Object Linking and Embedding (OLE) automation is part of the OLE 2.0 architecture from Microsoft Corporation. With OLE automation, your applications, regardless of the language in which they are written, can expose their properties and methods in OLE automation objects. Other applications, such as Lotus Notes® or Microsoft Exchange, can then integrate these objects by taking advantage of these properties and methods through OLE automation.

The applications exposing the properties and methods are called OLE automation servers or objects, and the applications that access those properties and methods are called OLE automation controllers. OLE automation servers are COM components (objects) that implement the OLE IDispatch interface. An OLE automation controller is a COM client that communicates with the automation server through its IDispatch interface. COM is the foundation of OLE. For OLE automation routines, DB2 acts as an OLE automation controller. Through this mechanism, DB2 can invoke methods of OLE automation objects as external routines.

Note that all OLE automation topics assume that you are familiar with OLE automation terms and concepts. For an overview of OLE automation, refer to *Microsoft Corporation: The Component Object Model Specification*, October 1995. For details on OLE automation, refer to *OLE Automation Programmer's Reference*, Microsoft Press, 1996, ISBN 1-55615-851-3.

Creating OLE automation routines

OLE automation routines are implemented as public methods of OLE automation objects. The OLE automation objects must be externally creatable by an OLE automation controller, in this case DB2, and support late binding (also called IDispatch-based binding). OLE automation objects must be registered in the Windows registry with a class identifier (CLSID), and optionally, an OLE programmatic ID (progID) to identify the automation object. The progID can identify an in-process (.DLL) or local (.EXE) OLE automation server, or a remote server through DCOM (Distributed COM).

To register OLE automation routines:

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

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

The external name consists of the OLE progID identifying the OLE automation object and the method name separated by ! (exclamation mark):

```
CREATE FUNCTION bcounter () RETURNS INTEGER
  EXTERNAL NAME 'bert.bcounter!increment'
  LANGUAGE OLE
  FENCED
  NOT THREADSAFE
```

© Copyright IBM Corp. 1993, 2008

```
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 (methods of OLE automation objects) are applied on instances of OLE automation objects. DB2 creates an object instance for each UDF or method reference in an SQL statement. An object instance can be reused for subsequent method invocations of the UDF or method reference in an SQL statement, or the instance can be released after the method invocation and a new instance is created for each subsequent method invocation. The proper behavior can be specified with the SCRATCHPAD option in the CREATE statement. For the LANGUAGE OLE clause, the SCRATCHPAD option has the additional semantic compared to C or C++, that a single object instance is created and reused for the entire query, whereas if NO SCRATCHPAD is specified, a new object instance can be created each time a method is invoked.

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

Supported SQL data types in OLE automation

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

Table 44. Mapping of SQL and OLE Automation Datatypes

SQL Type	OLE Automation Type	OLE Automation Type Description
SMALLINT	short	16-bit signed integer
INTEGER	long	32-bit signed integer
REAL	float	32-bit IEEE floating-point number
FLOAT or DOUBLE	double	64-bit IEEE floating-point number
DATE	DATE	64-bit floating-point fractional number of days since December 30, 1899
TIME	DATE	
TIMESTAMP	DATE	

Table 44. Mapping of SQL and OLE Automation Datatypes (continued)

SQL Type	OLE Automation Type	OLE Automation Type Description
CHAR(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARCHAR(*n*)	BSTR	
LONG VARCHAR	BSTR	
CLOB(*n*)	BSTR	
GRAPHIC(*n*)	BSTR	Length-prefixed string as described in the *OLE Automation Programmer's Reference*.
VARGRAPHIC(*n*)	BSTR	
LONG GRAPHIC	BSTR	
DBCLOB(*n*)	BSTR	
CHAR(*n*)	SAFEARRAY[unsigned char]	1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the *OLE Automation Programmer's Reference*.)
VARCHAR(*n*)	SAFEARRAY[unsigned char]	
LONG VARCHAR	SAFEARRAY[unsigned char]	
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	
BLOB(*n*)	SAFEARRAY[unsigned char]	

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

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

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

Table 45. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types (continued)

SQL Type	OLE Automation Type	BASIC Type	C++ Type
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
LONG VARCHAR	BSTR	String	BSTR
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
BLOB(*n*)	BSTR	String	BSTR
BLOB(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
GRAPHIC(*n*), VARGRAPHIC(*n*), LONG GRAPHIC, DBCLOB(*n*)	BSTR	String	BSTR

OLE Automation in BASIC

To implement OLE automation routines in BASIC you need to use the BASIC data types corresponding to the SQL data types mapped to OLE automation types.

The BASIC declaration of the OLE automation UDF, `bcounter`, looks like the following:

```
Public Sub increment(output As Long, _
                     indicator As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                     sqlmsg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)
```

OLE Automation in C++

The C++ declaration of the OLE automation UDF, `increment`, is as follows:

```
STDMETHODIMP Ccounter::increment (long   *output,
                                 short  *indicator,
                                 BSTR   *sqlstate,
                                 BSTR   *fname,
                                 BSTR   *fspecname,
                                 BSTR   *sqlmsg,
                                 SAFEARRAY **scratchpad,
                                 long   *calltype );
```

OLE supports type libraries that describe the properties and methods of OLE automation objects. Exposed objects, properties, and methods are described in the Object Description Language (ODL). The ODL description of the above C++ method is as follows:

```
HRESULT increment ([out]    long  *output,
                   [out]    short *indicator,
                   [out]    BSTR  *sqlstate,
                   [in]     BSTR  *fname,
                   [in]     BSTR  *fspecname,
                   [out]    BSTR  *sqlmsg,
                   [in,out] SAFEARRAY (unsigned char) *scratchpad,
                   [in]     long *calltype);
```

The ODL description allows you to specify whether a parameter is an input (in), output (out), or input/output (in,out) parameter. For an OLE automation routine, the routine input parameters and input indicators are specified as [in] parameters, and routine output parameters and output indicators as [out] parameters. For the

routine trailing arguments, `sqlstate` is an [out] parameter, `fname` and `fspecname` are [in] parameters, `scratchpad` is an [in,out] parameter, and `calltype` is an [in] parameter.

OLE automation defines the BSTR data type to handle strings. BSTR is defined as a pointer to OLECHAR: `typedef OLECHAR *BSTR`. For allocating and freeing BSTRs, OLE imposes the rule that the called routine frees a BSTR passed in as a by-reference parameter before assigning the parameter a new value. The same rule applies for one-dimensional byte arrays that are received by the called routine as `SAFEARRAY**`. This rule means the following for DB2 and OLE automation routines:

- [in] parameters: DB2 allocates and frees [in] parameters.
- [out] parameters: DB2 passes in a pointer to NULL. The [out] parameter must be allocated by the invoked routine and is freed by DB2.
- [in,out] parameters: DB2 initially allocates [in,out] parameters. They can be freed and re-allocated by the invoked routine. As is true for [out] parameters, DB2 frees the final returned parameter.

All other parameters are passed as pointers. DB2 allocates and manages the referenced memory.

OLE automation provides a set of data manipulation functions for dealing with BSTRs and SAFEARRAYs. The data manipulation functions are described in the *OLE Automation Programmer's Reference*.

The following C++ routine returns the first 5 characters of a CLOB input parameter:

```
// UDF DDL: CREATE FUNCTION crunch (CLOB(5k)) RETURNS CHAR(5)

STDMETHODIMP Cobj::crunch (BSTR *in,          // CLOB(5K)
                           BSTR *out,         // CHAR(5)
                           short *indicator1,  // input indicator
                           short *indicator2,  // output indicator
                           BSTR *sqlstate,    // pointer to NULL
                           BSTR *fname,       // pointer to function name
                           BSTR *fspecname,   // pointer to specific name
                           BSTR *msgtext)     // pointer to NULL
  {
     // Allocate BSTR of 5 characters
     // and copy 5 characters of input parameter

     // out is an [out] parameter of type BSTR, that is,
     // it is a pointer to NULL and the memory does not have to be freed.
     // DB2 will free the allocated BSTR.

     *out = SysAllocStringLen (*in, 5);
     return NOERROR;
  };
```

An OLE automation server can be implemented as *creatable single-use* or *creatable multi-use*. With creatable single-use, each client (that is, a DB2 FENCED process) connecting with `CoGetClassObject` to an OLE automation object will use its own instance of a class factory, and run a new copy of the OLE automation server if necessary. With creatable multi-use, many clients connect to the same class factory. That is, each instantiation of a class factory is supplied by an already running copy of the OLE server, if any. If there are no copies of the OLE server running, a copy is automatically started to supply the class object. The choice between single-use and multi-use OLE automation servers is yours, when you implement your automation server. A single-use server is recommended for better performance.

Chapter 10. 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 DB2 database system simplifies the creation of OLE DB applications by enabling you to define table functions that access an OLE DB data source. DB2 is an OLE DB consumer that can access any OLE DB data or service provider. You can perform operations including GROUP BY, JOIN, and UNION on data sources that expose their data through OLE DB interfaces. For example, you can define an OLE DB table function to return a table from a Microsoft Access database or a Microsoft Exchange address book, then create a report that seamlessly combines data from this OLE DB table function with data in your DB2 database.

Using OLE DB table functions reduces your application development effort by providing built-in access to any OLE DB provider. For C, Java, and OLE automation table functions, the developer needs to implement the table function, whereas in the case of OLE DB table functions, a generic built-in OLE DB consumer interfaces with any OLE DB provider to retrieve data. You only need to register a table function as LANGUAGE OLEDB, and refer to the OLE DB provider and the relevant rowset as a data source. You do not have to do any UDF programming to take advantage of OLE DB table functions.

To use OLE DB table functions with the DB2 database, you must install OLE DB 2.0 or later, available from Microsoft at `http://www.microsoft.com`. If you attempt to invoke an OLE DB table function without first installing OLE DB, DB2 issues SQLCODE -465, SQLSTATE 58032, reason code 35. For the system requirements and OLE DB providers available for your data sources, refer to your data source documentation. For the OLE DB specification, see the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Restrictions on using OLE DB table functions: OLE DB table functions cannot connect to a DB2 database.

Creating an OLE DB table UDF

To define an OLE DB table function with a single CREATE FUNCTION statement, you must:

- define the table that the OLE DB provider returns
- specify LANGUAGE OLEDB
- identify the OLE DB rowset and provide an OLE DB provider connection string in the EXTERNAL NAME clause

OLE DB data sources expose their data in tabular form, called a *rowset*. A rowset is a set of rows, each having a set of columns. The RETURNS TABLE clause includes

© IBM Corporation 1993, 2008

only the columns relevant to the user. The binding of table function columns to columns of a rowset at an OLE DB data source is based on column names. If the OLE DB provider is case sensitive, place the column names in quotation marks; for example, "UPPERcase".

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

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

where:

server identifies a server registered with the CREATE SERVER statement

rowset identifies a rowset, or table, exposed by the OLE DB provider; this value should be empty if the table has an input parameter to pass through command text to the OLE DB provider.

connectstring
contains initialization properties needed to connect to an OLE DB provider. For the complete syntax and semantics of the connection string, see the "Data Link API of the OLE DB Core Components" in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

You can use a *connection string* in the EXTERNAL NAME clause of a CREATE FUNCTION statement, or specify the *CONNECTSTRING* option in a CREATE SERVER statement.

For example, you can define an OLE DB table function and return a table from a Microsoft Access database with the following CREATE FUNCTION and SELECT statements:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
                 Data Source=c:\msdasdk\bin\oledb\nwind.mdb';

SELECT orderid, DATE(orderdate) AS orderdate,
                DATE(shippeddate) AS shippeddate
FROM TABLE(orders()) AS t
WHERE orderid = 10248;
```

Instead of putting the connection string in the EXTERNAL NAME clause, you can create and use a server name. For example, assuming you have defined the server Nwind, you could use the following CREATE FUNCTION statement:

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

OLE DB table functions also allow you to specify one input parameter of any character string data type. Use the input parameter to pass command text directly to the OLE DB provider. If you define an input parameter, do not provide a rowset name in the EXTERNAL NAME clause. DB2 passes the command text to the OLE DB provider for execution and the OLE DB provider returns a rowset to DB2. Column names and data types of the resulting rowset need to be compatible with the RETURNS TABLE definition in the CREATE FUNCTION statement. Since binding to the column names of the rowset is based on matching column names, you must ensure that you name the columns properly.

The following example registers an OLE DB table function, which retrieves store information from a Microsoft SQL Server 7.0 database. The connection string is provided in the EXTERNAL NAME clause. Since the table function has an input parameter to pass through command text to the OLE DB provider, the rowset name is not specified in the EXTERNAL NAME clause. The query example passes in a SQL command text that retrieves information about the top three stores from a SQL Server database.

```
CREATE FUNCTION favorites (VARCHAR(600))
  RETURNS TABLE (store_id CHAR (4), name VARCHAR (41), sales INTEGER)
  SPECIFIC favorites
  LANGUAGE OLEDB
  EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
  User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
  Locale Identifier=1033;Use Procedure for Prepare=1;
  Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
  OLE DB Services=CLIENTCURSOR;';

SELECT *
FROM TABLE (favorites (' select top 3 sales.stor_id as store_id,  ' ||
                       '        stores.stor_name as name,         ' ||
                       '        sum(sales. qty) as sales          ' ||
                       ' from sales, stores                       ' ||
                       ' where sales.stor_id = stores.stor_id     ' ||
                       ' group by sales.stor_id, stores.stor_name ' ||
                       ' order by sum(sales.qty) desc')) as f;
```

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 following table shows how DB2 data types map to the OLE DB data types as described in *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. Use the mapping table to define the appropriate RETURNS TABLE columns in your OLE DB table functions. For example, if you define an OLE DB table function with a column of data type INTEGER, DB2 requests the data from the OLE DB provider as DBTYPE_I4.

For mappings of OLE DB provider source data types to OLE DB data types, refer to the OLE DB provider documentation. For examples of how the ANSI SQL, Microsoft Access, and Microsoft SQL Server providers might map their respective data types to OLE DB data types, refer to the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Table 46. Mapping DB2 Data Types to OLE DB

DB2 Data Type	OLE DB Data Type
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

Note: OLE DB data type conversion rules are defined in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. For example:

- To retrieve the OLE DB data type DBTYPE_CY, the data can get converted to OLE DB data type DBTYPE_NUMERIC(19,4), which maps to DB2 data type DEC(19,4).
- To retrieve the OLE DB data type DBTYPE_I1, the data can get converted to OLE DB data type DBTYPE_I2, which maps to DB2 data type SMALLINT.
- To retrieve the OLE DB data type DBTYPE_GUID, the data can get converted to OLE DB data type DBTYPE_BYTES, which maps to DB2 data type CHAR(12) FOR BIT DATA.

Chapter 11. Routine invocation

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.

© Copyright IBM Corp. 1993, 2008

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 341
- "Calling procedures from triggers or SQL routines" on page 342
- 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 345
- "Invoking scalar functions or methods" on page 352
- "Invoking user-defined table functions" on page 353

Authorizations and binding of routines that contain SQL

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

Package Owner

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

Routine Definer

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

authorization of the routine invoker). For a routine to be successfully invoked, the routine definer must have one of:

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

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

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

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

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

Routine Invoker

The ID that invokes the routine. To determine which users will be invokers of a routine, it is necessary to consider how a routine can be invoked. Routines can be invoked from a command window or from within an embedded SQL application. In the case of methods and UDFs the routine reference will be embedded in another SQL statement. A procedure is invoked by using the CALL statement. For dynamic SQL in an application, the invoker is the runtime authorization ID of the immediately higher-level routine or application containing the routine invocation (however, this ID can also depend on the DYNAMICRULES option with which the higher-level routine or application was bound). For static SQL, the invoker is the value of the OWNER precompile/BIND option of the package that contains the reference to the routine. To successfully invoke the routine, these users will require EXECUTE privilege on the routine. This privilege can be granted by any user with EXECUTE WITH GRANT OPTION privilege on the routine (this includes the routine definer unless the privilege has been explicitly revoked), SYSADM or DBADM authority by explicitly issuing a GRANT statement.

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

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:

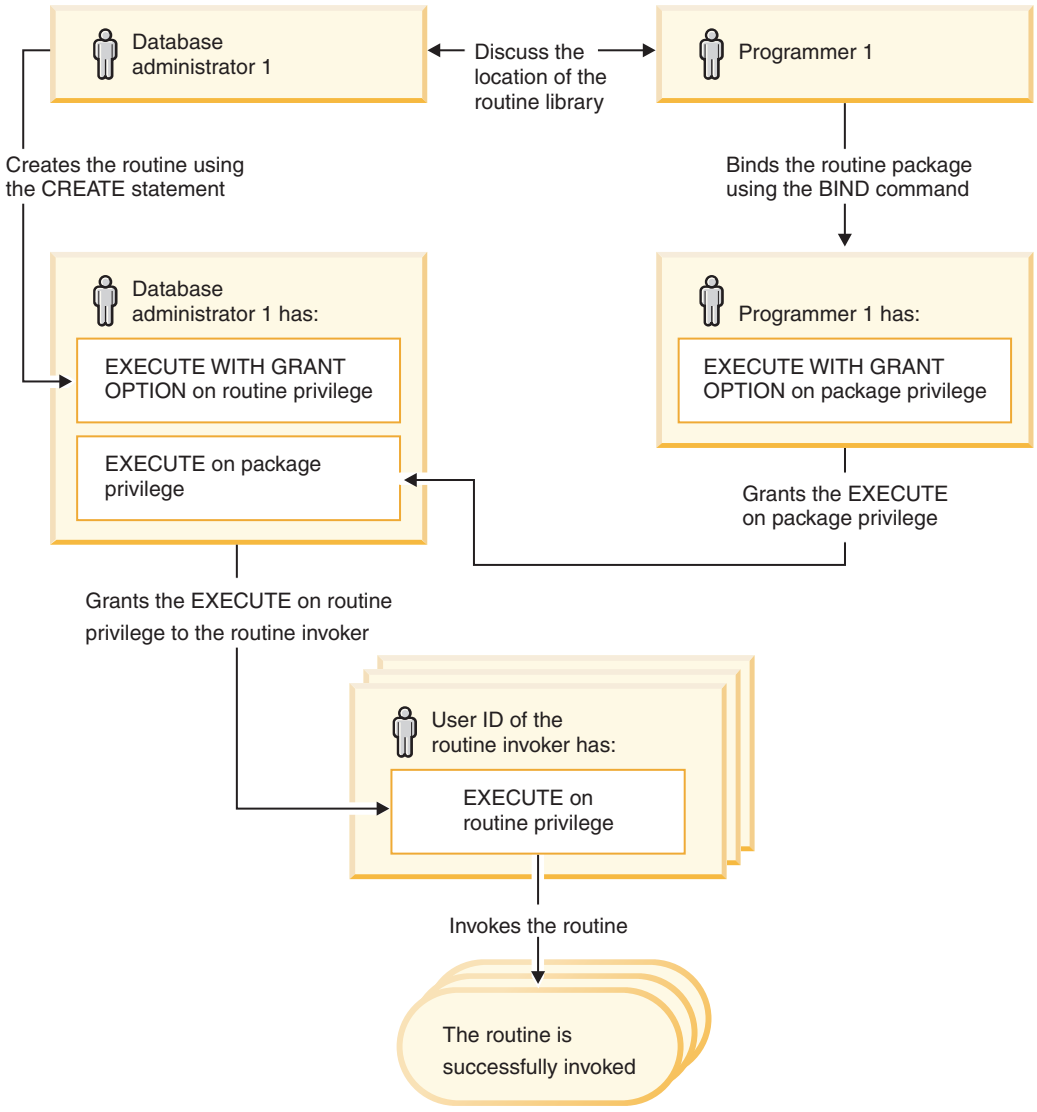


Figure 3. Managing the EXECUTE privilege on routines

1. Definer performs the appropriate CREATE statement to register the routine. This registers the routine in DB2 with its intended level of SQL access, establishes the routine signature, and also points to the routine executable. The definer, if not also the package owner, needs to communicate with the package owners and authors of the routine programs to be clear on where the routine libraries reside so that this can be correctly specified in the EXTERNAL clause of the CREATE statement. By virtue of a successful CREATE statement, the

definer has EXECUTE WITH GRANT privilege on the routine, however the definer does not yet have EXECUTE privilege on the packages of the routine.

2. Definer must grant EXECUTE privilege on the routine to any users who are to be permitted use of the routine. (If the package for this routine will recursively call this routine, then this step must be done before the next step.)
3. Package owners precompile and bind the routine program, or have it done on their behalf. Upon a successful precompile and bind, the package owner is implicitly granted EXECUTE WITH GRANT OPTION privilege on the respective package. This step follows step one in this list only to cover the possibility of SQL recursion in the routine. If such recursion does not exist in any particular case, the precompile/bind could precede the issuing of the CREATE statement for the routine.
4. Each package owner must explicitly grant EXECUTE privilege on their respective routine package to the definer of the routine. This step must come at some time after the previous step. If the package owner is also the routine definer, this step can be skipped.
5. Static usage of the routine: the bind owner of the package referencing the routine must have been given EXECUTE privilege on the routine, so the previous step must be completed at this point. When the routine executes, DB2 verifies that the definer has the EXECUTE privilege on any package that is needed, so step 3 must be completed for each such package.
6. Dynamic usage of the routine: the authorization ID as controlled by the DYNAMICRULES option for the invoking application must have EXECUTE privilege on the routine (step 4), and the definer of the routine must have the EXECUTE privilege on the packages (step 3).

Routine names and paths

The qualified name of a stored procedure or UDF is `schema-name.routine-name`. You can use this qualified name anywhere you refer to a stored procedure or UDF. For example:

```
SANDRA.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
```

However, you can also omit the `schema-name.`, in which case, DB2 will attempt to identify the stored procedure or UDF to which you are referring. For example:

```
BOAT_COMPARE    FOO    SUBSTR    FLOOR
```

The qualified name of a method is `schema-name.type..method-name`.

The concept of *SQL path* is central to DB2's resolution of *unqualified* references that occur when you do not use the `schema-name`. The SQL path is an ordered list of schema names. It provides a set of schemas for resolving unqualified references to stored procedures, UDFs, and types. In cases where a reference matches a stored procedure, type, or UDF in more than one schema in the path, the order of the schemas in the path is used to resolve this match. The SQL path is established by means of the FUNCPATH option on the precompile and bind commands for static SQL. The SQL path is set by the SET PATH statement for dynamic SQL. The SQL path has the following default value:

```
"SYSIBM","SYSFUN","SYSPROC", "ID"
```

This applies to both static and dynamic SQL, where *ID* represents the current statement authorization ID.

Routine names can be *overloaded*, which means that multiple routines, even in the same schema, can have the same name. Multiple functions or methods with the same name can have the same number of parameters, as long as the data types differ. This is not true for stored procedures, where multiple stored procedures with the same name must have different numbers of parameters. Instances of different routine types do not overload one-another, except for methods, which are able to overload functions. For a method to overload a function, the method must be registered using the WITH FUNCTION ACCESS clause.

A function, a stored procedure, and a method can have identical *signatures* and be in the same schema without overloading each other. In the context of routines, signatures are the qualified routine name concatenated with the defined data types of all the parameters in the order in which they are defined.

Methods are invoked against instances of their associated structured type. When a subtype is created, among the attributes it inherits are the methods defined for the supertype. Hence, a supertype's methods can also be run against any instances of its subtypes. When defining a subtype you can *override* the supertype's method. To override a method means to reimplement it specifically for a given subtype. This facilitates the dynamic dispatch of methods (also known as polymorphism), where an application will execute the most specific method depending on the type of the structured type instance (for example, where it is situated in the structured type hierarchy).

Each routine type has its own selection algorithm that takes into account the facts of overloading (in the case of methods, and overriding) and SQL path to choose the most appropriate match for every routine reference.

Nested routine invocations

In the context of routines, *nesting* refers to the situation where one routine invokes another. That is to say, the SQL issued by one routine can reference another routine, which could issue SQL that again references another routine, and so on. If the series of routines that is referenced contains a routine that was previously referenced this is said to be a *recursive* nesting situation.

You can use nesting and recursion in your DB2 routines under the following restrictions:

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 DB2 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. 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 DB2 instances is discouraged.

To successfully invoke Java procedures in a 64-bit DB2 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 DB2 instances. Since 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.

To invoke existing 32-bit routines on a 64-bit server:

1. Copy the routine class or library to the database routines directory:
   - UNIX: sqllib/function
   - Windows: sqllib\function
2. Register the stored procedure with the CREATE PROCEDURE statement.
3. Invoke the stored procedure with the CALL statement.

Routine code page considerations

Character data is passed to external routines in the code page implied by the PARAMETER CCSID option used when the routine was created. Similarly, a character string that is output from the routine is assumed by the database to use the code page implied by the PARAMETER CCSID option.

When a client program (using, for example, code page C) accesses a section with a different code page (for example, code page S) that invokes a routine using a different code page (for example, code page R), the following events occur:

1. When an SQL statement is invoked, input character data is converted from the code page of the client application (C) to the one associated with the section (S). Conversion does not occur for BLOBs or data that will be used as FOR BIT DATA.
2. If the code page of the routine is not the same as the code page of the section, then before the routine is invoked, input character data (except for BLOB and FOR BIT DATA) is converted to the code page of the routine (R).

   It is strongly recommended that you precompile, compile, and bind the server routine using the code page that the routine will be invoked under (R). This might not be possible in all cases. For example, you can create a Unicode database in a Windows environment. However, if the Windows environment does not have the Unicode code page, you have to precompile, compile, and

bind the application that creates the routine in a Windows code page. The routine will work if the application has no special delimiter characters that the precompiler does not understand.

3. When the routine finishes, the database manager converts all output character data from the routine code page (R) to the section code page (S) if necessary. If the routine raised an error during its execution, the SQLSTATE and diagnostic message from the routine will also be converted from the routine code page to the section code page. Conversion does not happen for BLOB or FOR BIT DATA character strings.
4. When the statement finishes, output character data is converted from the section code page (S) back to code page of the client application (C). Conversion does not occur for BLOBS or for data that was used as FOR BIT DATA.

By using the DBINFO option on the CREATE FUNCTION, CREATE PROCEDURE, and CREATE TYPE statements, the routine code page is passed to the routine. Using this information, a routine that is sensitive to the code page can be written to operate in many different code pages.

References to procedures

Stored Procedures are invoked from the CALL statement where they are referenced by a qualified name (schema and stored procedure name), followed by a list of arguments enclosed by parentheses. A stored procedure can also be invoked without the schema name, resulting in a choice of possible stored procedures in different schemas with the same number of parameters.

Each parameter passed to the stored procedure can be composed of a host variable, parameter marker, expression, or NULL. The following are restrictions for stored procedure parameters:

- OUT and INOUT parameters must be host variables.
- NULLs cannot be passed to Java stored procedures unless the SQL data type maps to a Java class type.
- NULLs cannot be passed to PARAMETER STYLE GENERAL stored procedures.

The position of the arguments is important and must conform to the stored procedure definition for the semantics to be correct. Both the position of the arguments and the stored procedure definition must conform to the stored procedure body itself. DB2 does not attempt to shuffle arguments to better match a stored procedure definition, and DB2 does not understand the semantics of the individual stored procedure parameters.

Procedure selection

Given a stored procedure invocation, the database manager must decide which of the possible stored procedures with the same name to call. Stored procedure resolution is done using the steps that follow.

1. Find all stored procedures from the catalog (SYSCAT.ROUTINES), such that all of the following are true:
   - For invocations where the schema name was specified (that is, qualified references), the schema name and the stored procedure name match the invocation name.

- For invocations where the schema name was not specified (that is, unqualified references), the stored procedure name matches the invocation name, and has a schema name that matches one of the schemas in the SQL path.
- The number of defined parameters matches the invocation.
- The invoker has the EXECUTE privilege on the stored procedure.

2. Choose the stored procedure whose schema is earliest in the SQL path.

If there are no candidate stored procedures remaining after the first step, an error is returned (SQLSTATE 42884).

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. 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 platform can use the CALL statement to invoke a procedure residing on a Linux database server.

To see how you can call a procedure from a particular interface see:

- "Calling procedures from triggers or SQL routines" on page 342
- "Calling procedures from applications or external routines"
- "Calling stored procedures from CLI applications" in *Call Level Interface Guide and Reference, Volume 1*
- "Calling procedures from the Command Line Processor (CLP)" on page 345

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.

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.

Certain elements must be included in your application if you want that application to invoke a procedure. In writing your application you must do the following:

1. Declare, allocate, and initialize storage for the optional data structures and host variables or parameter markers required for the CALL statement. To do this:
   - Assign a host variable or parameter marker to be used for each parameter of the procedure.
   - Initialize the host variables or parameter markers that correspond to IN or INOUT parameters.
2. Establish a database connection. Do this by executing an embedded SQL language CONNECT TO statement, or by coding an implicit database connection.
3. Code the procedure invocation. After the database connection code, you can code the procedure invocation. Do this by executing the SQL language CALL statement. Be sure to specify a host variable, constant, or parameter marker for each IN, INOUT, OUT parameter that the procedure expects.
4. Add code to process the OUT and INOUT parameters, and result sets. This code must come after the CALL statement execution.
5. Code a database COMMIT or ROLLBACK. Subsequent to the CALL statement and evaluation of output parameter values or data returned by the procedure, you might want your application to commit or roll back the transaction. This can be done by including a COMMIT or ROLLBACK statement. A procedure can include a COMMIT or ROLLBACK statement, however it is recommended practice that transaction management be done within the client application.

   Note: Procedures invoked from an application that established a type 2 connection to the database, cannot issue COMMIT or ROLLBACK statements.
6. Disconnect from the database.
7. Prepare, compile, link, and bind your application. If the application is for an external routine, issue the CREATE statement to create the routine and locate your external code library in the appropriate function path for your operating system so that the database manager can find it.
8. Run your application or invoke your external routine. The CALL statement that you embedded in your application will be invoked.

Note: You can code SQL statements and routine logic at any point between steps 2 and 5.

Calling procedures from triggers or SQL routines

Calling a procedure from an SQL routine, a trigger, or dynamic compound statement is essentially the same. The same steps are used to implement this call. This topic explains the steps using a trigger scenario. Any prerequisites or steps that differ when calling a procedure from a routine or dynamic compound statement are stated.

Prerequisites

- The procedure must have been created in the database by executing the CREATE PROCEDURE statement.
- For external procedures, the library or class files must be in the location specified by the EXTERNAL clause of the CREATE PROCEDURE statement.
- The creator of a trigger that contains a CALL statement must have the privilege to execute the CALL statement. At runtime when a trigger is activated it is the authorization of the creator of the trigger that is checked for the privilege to execute the CALL statement. A user that executes a dynamic compound statement that contains a CALL statement, must have the privilege to execute the CALL statement for that procedure.
- To invoke a trigger, a user must have the privilege to execute the data change statement associated with the trigger event. Similarly, to successfully invoke an SQL routine or dynamic compound statement a user must have the EXECUTE privilege on the routine.

When invoking a procedure from within an SQL trigger, an SQL routine, or a dynamic compound statement the following restrictions apply:

- In partitioned database environments procedures cannot be invoked from triggers or SQL UDFs.
- On symmetric multi-processor (SMP) machines, procedure calls from triggers are executed on a single processor.
- A procedure that is to be called from a trigger must not contain a COMMIT statement or a ROLLBACK statement that attempts to rollback the unit of work. The ROLLBACK TO SAVEPOINT statement is supported within the procedure however the specified savepoint must be in the procedure.
- A rollback of a CALL statement from within a trigger will not rollback 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.

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 40
- "SQL statements that can be executed in routines" on page 35

This procedure section explains how to create and invoke a trigger that contains a CALL statement. The SQL required to call a procedure from a trigger is the same SQL required to call a procedure from an SQL routine or dynamic compound statement.

1. Write a basic CREATE TRIGGER statement specifying the desired trigger attributes. See the CREATE TRIGGER statement.
2. In the trigger action portion of the trigger you can declare SQL variables for any IN, INOUT, OUT parameters that the procedure specifies. See the

DECLARE statement. To see how to initialize or set these variables see the assignment statement. Trigger transition variables can also be used as parameters to a procedure.

3. In the trigger action portion of the trigger add a CALL statement for the procedure. Specify a value or expression for each of the procedure's IN, INOUT, and OUT parameters
4. For SQL procedures you can optionally capture the return status of the procedure by using the GET DIAGNOSTICS statement. To do this you will need to use an integer type variable to hold the return status. Immediately after the CALL statement, simply add a GET DIAGNOSTICS statement that assigns RETURN_STATUS to your local trigger return status variable.
5. Having completed writing your CREATE TRIGGER statement you can now execute it statically (from within an application) or dynamically (from the CLP, or from the Control Center) to formally create the trigger in the database.
6. Invoke your trigger. Do this by executing against the appropriate data change statement that corresponds to your trigger event.
7. When the data change statement is executed against the table, the appropriate triggers defined for that table are fired. When the trigger action is executed, the SQL statements contained within it, including the CALL statement, are executed.

Run-time errors

If the procedure attempts to read or write to a table that the trigger also reads or writes to, an error might be raised if a read or write conflict is detected. The set of tables that the trigger modifies, including the table for which the trigger was defined must be exclusive from the tables modified by the procedure.

Example: Calling an SQL procedure from a trigger

This example illustrates how you can embed a CALL statement to invoke a procedure within a trigger and how to capture the return status of the procedure call using the GET DIAGNOSTICS statement. The SQL below creates the 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 PROCEDURE proc(IN val INT, IN name CHAR(2))
LANGUAGE SQL
DYNAMIC RESULTSETS 0
MODIFIES SQL DATA
BEGIN
  DECLARE rc INT DEFAULT 0;
  INSERT INTO TABLE T2 VALUES (val, name);
  GET DIAGNOSTICS rc = ROW_COUNT;
  IF ( rc > 0 ) THEN
      RETURN 0;
  ELSE
      RETURN -200;
  END IF;
END@

CREATE TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.c1 > 100);
BEGIN ATOMIC
   DECLARE rs INTEGER DEFAULT 0;
```

```
   CALL proc(n.c1, n.c2);
   GET DIAGNOSTICS rs = RETURN_STATUS;
   VALUES(CASE WHEN rc < 0 THEN RAISE_ERROR('70001', 'PROC CALL failed'));
END@
```

Issuing the following SQL statement will cause the trigger to fire and the procedure will be invoked.

```
UPDATE T1 SET c1 = c1+1 WHERE c2 = 'CA'@
```

Calling procedures from the Command Line Processor (CLP)

You can call stored procedures by using the CALL statement from the DB2 command line processor interface. The stored procedure being called must be defined in the DB2 system catalog tables.

To call the stored procedure, first connect to the database:

```
db2 connect to sample user userid using password
```

where *userid* and *password* are the user ID and password of the instance where the sample database is located.

To use the CALL statement, enter the stored procedure name plus any IN or INOUT parameter values, as well as '?' as a place-holder for each OUT parameter value.

The parameters for a stored procedure are given in the CREATE PROCEDURE statement for the stored procedure in the program source file.

SQL procedure 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)
```

To invoke this procedure, use the CALL statement in which you must specify the procedure name and appropriate parameter arguments, which in this case are the value for the IN parameter, and a question mark, '?', for the value of the OUT parameter. The procedure's SELECT statement uses the deptNumber value on the DEPT column of the STAFF table, so to get meaningful output the IN parameter needs to be a valid value from the DEPT column; for example, the value "51":

```
db2 call dept_median (51, ?)
```

Note: On UNIX platforms the parentheses have special meaning to the command shell, so they must be preceded with a "\" character or surrounded with 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.
After running the above command, you should receive this result:

```
Value of output parameters
--------------------------
Parameter Name  : MEDIANSALARY
Parameter Value : +1.76545000000000E+004

Return Status = 0
```

Example 2.

This example illustrates how to call a procedure with array parameters. Type phonenumbers is defined as:

```
CREATE TYPE phonenumbers AS VARCHAR(12) ARRAY[1000]
```

Procedure find_customers, defined below, has an IN and an OUT parameter of type phonenumbers. The procedure searches for numbers in numbers_in that begin with the given area_code, and reports them in numbers_out.

```
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 invoke the 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, when a procedure has an input parameter of an array data type, the input argument can be specified with an array constructor containing a list of literal values.

After running the command, you should receive a result like this:

```
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 created from supported host languages with the Command Line Processor. In the `samples/c` directory on UNIX, and the `samples\c` directory on Windows, DB2 provides files for creating stored procedures. The `spserver` shared library contains a number of stored procedures that can be created from the source file, `spserver.sqc`. The `spcreate.db2` file catalogs the stored procedures.

In the `spcreate.db2` file, the `CREATE PROCEDURE` statement for the `MAIN_EXAMPLE` procedure begins:

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
                               OUT salary DOUBLE,
                               OUT errorcode INTEGER)
```

To call this stored procedure, you need to put in a CHAR value for the IN parameter, `job`, and a question mark, '?', for each of the OUT parameters. The procedure's SELECT statement uses the `job` value on the JOB column of the EMPLOYEE table, so to get meaningful output the IN parameter needs to be a valid value from the JOB column. The C sample program, `spclient`, that calls the stored procedure, uses `'DESIGNER'` for the `JOB` value. We can do the same, as follows:

```
db2 "call MAIN_EXAMPLE ('DESIGNER', ?, ?)"
```

After running the above command, you should receive this result:

```
Value of output parameters
--------------------------
Parameter Name  : SALARY
Parameter Value : +2.37312500000000E+004

Parameter Name  : ERRORCODE
Parameter Value : 0

Return Status = 0
```

An `ERRORCODE` of zero indicates a successful result.

Comparing with the `spclient` program, notice that `spclient` has formatted the result in decimal for easier viewing:

```
CALL stored procedure named MAIN_EXAMPLE
Stored procedure returned successfully
Average salary for job DESIGNER =  23731.25
```

Calling stored procedures from CLI applications

CLI applications invoke stored procedures by executing the CALL procedure SQL statement. This topic describes how to call stored procedures from CLI applications.

Before calling a stored procedure, ensure that you have initialized your CLI application.

If the stored procedure being called is uncataloged, ensure that it does not call any of the CLI schema functions. Calling CLI schema functions from uncataloged stored procedures is not supported.

The CLI schema functions are: `SQLColumns()`, `SQLColumnPrivileges()`, `SQLForeignKeys()`, `SQLPrimaryKeys()`, `SQLProcedureColumns()`, `SQLProcedures()`, `SQLSpecialColumns()`, `SQLStatistics()`, `SQLTables()`, and `SQLTablePrivileges()`.

To call a stored procedure:

1. Declare application host variables corresponding to each of the IN, INOUT, and OUT parameters of the stored procedure. Ensure the application variable data types and lengths match the data types and lengths of the arguments in the stored procedure signature. DB2 CLI supports calling stored procedures with all SQL types as parameter markers.
2. Initialize the IN, INOUT, and OUT parameter application variables.
3. Issue the CALL SQL statement. For example:

```
SQLCHAR *stmt = (SQLCHAR *)"CALL OUT_LANGUAGE (?)";
```

For optimal performance, applications should use parameter markers for stored procedure arguments in the CALL procedure string and then bind the host variables to those parameter markers. If inbound stored procedure arguments must be specified as string literals rather than parameter markers, however, include the ODBC call escape clause delimiters { } in the CALL procedure statement. For example:

```
SQLCHAR *stmt = (SQLCHAR *)"{CALL IN_PARAM (123, 'Hello World!')}";
```

When string literals and the ODBC escape clause are used in a CALL procedure statement, the string literals can only be specified as IN mode stored procedure arguments. INOUT and OUT mode stored procedure arguments must still be specified using parameter markers.

4. Optional: Prepare the CALL statement by calling `SQLPrepare()`.
5. Bind each parameter of the CALL procedure statement by calling `SQLBindParameter()`.

   Note: Ensure each parameter is bound correctly (to SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT), otherwise unexpected results could occur when the CALL procedure statement is executed. This would happen, for example, if an input parameter was incorrectly bound with an *InputOutputType* of SQL_PARAM_OUTPUT.
6. Execute the CALL procedure statement using `SQLExecDirect()`, or if the CALL procedure statement was prepared in step 4, `SQLExecute()`.

   Note: If an application or thread that has invoked a stored procedure is terminated before the stored procedure completes, execution of the stored procedure will also be terminated. It is important that a stored procedure contain logic to ensure that the database is in both a consistent and desirable state if the stored procedure is terminated prematurely.
7. Check the return code of `SQLExecDirect()` or `SQLExecute()` when the function has returned to determine if any errors occurred during execution of either the CALL procedure statement or the stored procedure. If the return code is SQL_SUCCESS_WITH_INFO or SQL_ERROR, use the CLI diagnostic functions `SQLGetDiagRec()` and `SQLGetDiagField()` to determine why the error occurred.

   If a stored procedure has executed successfully, any variables bound as OUT parameters may contain data that the stored procedure has passed back to the CLI application. If applicable, the stored procedure may also return one or more result sets through non-scrollable cursors. CLI applications should process stored procedure result sets as they would process result sets generated by executing SELECT statements.

   Note: If a CLI application is unsure of the number or type of parameters in a result set returned by a stored procedure, the `SQLNumResultCols()`, `SQLDescribeCol()`, and `SQLColAttribute()` functions can be called (in this order) on the result set to determine this information.

Once you have executed the CALL statement, you can retrieve result sets from the stored procedure if applicable.

Note:

The numeric month and day parts of a DATETYPE data type value will appear to be reversed in procedure result sets that are returned to DB2 CLI applications if the values are not returned in ISO format. For example, this can happen if a local format is used instead. To ensure that DATETYPE data type value information is

correctly intepreted by a client application, the procedures should be bound to the database with a locale-independent DATETIME format such as ISO. For example:

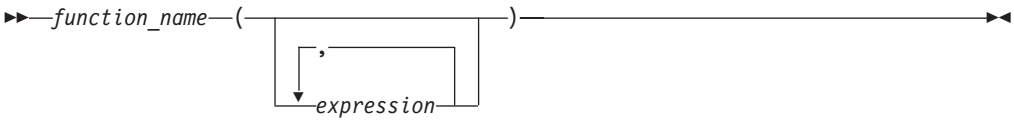
•
```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
```

Note:

DB2 CLI packages are automatically bound to databases when the databases are created or migrated.

References to functions

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:



In the preceding syntax diagram, `function_name` can be either an unqualified or a qualified function name. The arguments can number from 0 to 90 and are expressions. Examples of some components that can compose expressions are the following:

- a column name, qualified or unqualified
- a constant
- a host variable
- a special register
- a parameter marker

The position of the arguments is important and must conform to the function definition for the semantics to be correct. Both the position of the arguments and the function definition must conform to the function body itself. DB2 does not attempt to shuffle arguments to better match a function definition, and DB2 does not understand the semantics of the individual function parameters.

Use of column names in UDF argument expressions requires that the table references that contain the columns have proper scope. For table functions referenced in a join and using any argument involving columns from another table or table function, the referenced table or table function must precede the table function containing the reference in the FROM clause.

In order to use parameter markers in functions you cannot simply code the following:

```
BLOOP(?)
```

Because the function selection logic does not know what data type the argument might turn out to be, it cannot resolve the reference. You can use the CAST specification to provide a type for the parameter marker. For example, INTEGER, and then the function selection logic can proceed:

```
BLOOP(CAST(? AS INTEGER))
```

Some valid examples of function invocations are:

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT(SELECT SUM(length*length)
     FROM triangles
     WHERE id= 'J522'
     AND legtype <> 'HYP')
```

If any of the above functions are table functions, the syntax to reference them is slightly different than presented previously. For example, if PABLO.BLOOP is a table function, to properly reference it, use:

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

Function selection

For both qualified and unqualified function references, the function selection algorithm looks at all the *applicable* functions, both built-in and user-defined, that have:

- The given name
- The same number of defined parameters as arguments in the function reference
- Each parameter identical to or promotable from the type of the corresponding argument.

Applicable functions are functions in the named schema for a qualified reference, or functions in the schemas of the SQL path for an unqualified reference. The algorithm looks for an exact match, or failing that, a best match among these functions. The SQL path is used, in the case of an unqualified reference only, as the deciding factor if two identically good matches are found in different schemas.

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
```

If column1 is a DECIMAL or DOUBLE column, the inner BLOOP reference resolves to the second BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP resolves to the first BLOOP.

Alternatively, if column1 is a SMALLINT or INTEGER column, the inner bloop reference resolves to the first BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP also resolves to the first BLOOP. In this case, you are seeing nested references to the same function.

By defining a function with the name of one of the SQL operators, you can actually invoke a UDF using *infix notation*. For example, suppose you can attach some meaning to the "+" operator for values which have distinct type BOAT. You can define the following UDF:

```
CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
```

Then you can write the following valid SQL statement:

```
SELECT BOAT_COL1 + BOAT_COL2
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

But you can also write the equally valid statement:

```
SELECT "+"(BOAT_COL1, BOAT_COL2)
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

Note that you are not permitted to overload the built-in conditional operators such as `>`, `=`, `LIKE`, `IN`, and so on, in this way.

For a more thorough description of function selection, see the Function References section in the Functions topic listed in the related links.

Distinct types as UDF or method parameters

UDFs and methods can be defined with distinct types as parameters or as the result. DB2 will pass the value to the UDF or method in the format of the source data type of the distinct type.

Distinct type values that originate in a host variable and which are used as arguments to a UDF that has its corresponding parameter defined as a distinct type, **must be explicitly cast to the distinct type by the user**. There is no host language type for distinct types. DB2's strong typing necessitates this, otherwise your results can be ambiguous. Consider the BOAT distinct type which is defined over a BLOB, and consider the BOAT_COST UDF defined as follows:

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  ...
```

In the following fragment of a C language application, the host variable `:ship` holds the BLOB value that is to passed to the BOAT_COST function:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS BLOB(150K) ship;
EXEC SQL END DECLARE SECTION;
```

Both of the following statements correctly resolve to the BOAT_COST function, because both cast the `:ship` host variable to type BOAT:

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

If there are multiple BOAT distinct types in the database, or BOAT UDFs in other schema, you must exercise care with your SQL path. Your results can otherwise be ambiguous.

LOB values as UDF parameters

UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. DB2 will materialize the entire LOB value in storage before invoking such a function, even if the source of the value is a *LOB locator* host variable. For example, consider the following fragment of a C language application:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(150K) clob150K ;        /* LOB host var */
  SQL TYPE IS CLOB_LOCATOR clob_locator1;  /* LOB locator host var */
  char                   string[40];       /* string host var */
EXEC SQL END DECLARE SECTION;
```

Either host variable `:clob150K` or `:clob_locator1` is valid as an argument for a function whose corresponding parameter is defined as `CLOB(500K)`. For example, suppose you have registered a UDF as follows:

```
CREATE FUNCTION FINDSTRING (CLOB(500K, VARCHAR(200)))
  ...
```

Both of the following invocations of FINDSTRING are valid in the program:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

UDF parameters or results which have one of the LOB types can be created with the AS LOCATOR modifier. In this case, the entire LOB value is not materialized prior to invocation. Instead, a LOB LOCATOR is passed to the UDF, which can then use SQL to manipulate the actual bytes of the LOB value.

You can also use this capability on UDF parameters or results which have a distinct type that is based on a LOB. Note that the argument to such a function can be any LOB value of the defined type; it does not have to be a host variable defined as one of the LOCATOR types. The use of host variable locators as arguments is completely orthogonal to the use of AS LOCATOR in UDF parameters and result definitions.

Invoking scalar functions or methods

The invocation of built-in scalar functions, user-defined scalar-functions and methods is very similar. Scalar functions and methods can only be invoked where expressions are supported within an SQL statement.

Prerequisites

- For built-in functions, SYSIBM must be in the CURRENT PATH special register. SYSIBM is in CURRENT PATH by default.
- For user-defined scalar functions, the function must have been created in the database using either the CREATE FUNCTION or CREATE METHOD statement.
- For external user-defined scalar functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION or CREATE METHOD statement.

- To invoke a user-defined function or method, a user must have EXECUTE privilege on the function or method. If the function or method is to be used by all users, the EXECUTE privilege on the function or method can be granted to PUBLIC. For more privilege related information see the specific CREATE statement reference.

To invoke a scalar UDF or method:

Include a reference to it within an expression contained in an SQL statement where it is to process one or more input values. Functions and methods can be invoked anywhere that an expression is valid. Examples of where a scalar UDF or method can be referenced include the select-list of a query or in a VALUES clause.

For example, suppose that you have created a user-defined scalar function called TOTAL_SAL that adds the base salary and bonus together for each employee row in the EMPLOYEE table.

```
CREATE FUNCTION TOTAL_SAL
  (SALARY DECIMAL(9,2), BONUS DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SALARY+BONUS
```

The following is a SELECT statement that makes use of TOTAL_SAL:

```
SELECT LASTNAME, TOTAL_SAL(SALARY, BONUS) AS TOTAL
  FROM EMPLOYEE
```

Invoking user-defined table functions

Once the user-defined table function is written and registered with the database, you can invoke it in the FROM clause of a SELECT statement.

Prerequisites

- The table function must have been created in the database by executing the CREATE FUNCTION.
- For external user-defined table functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION.
- To invoke a user-defined table function a user must have EXECUTE privilege on the function. For more privilege related information see the CREATE FUNCTION reference.

Restrictions

For restrictions on invoking user-defined table functions, see the CREATE FUNCTION topics in the related links.

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
```

Appendix A. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command line help
  - Command help
  - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com®.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order. English DB2 Version 9.5 manuals in PDF format and translated versions can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg2700947.

Although the tables identify books available in print, the books might not be available in your country or region.

© IBM Corporation 1993, 2008

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

Note: The DB2 Information Center is updated more frequently than either the PDF or the hard-copy books.

Table 47. DB2 technical information

Name	Form Number	Available in print
Administrative API Reference	SC23-5842-01	Yes
Administrative Routines and Views	SC23-5843-01	No
Call Level Interface Guide and Reference, Volume 1	SC23-5844-01	Yes
Call Level Interface Guide and Reference, Volume 2	SC23-5845-01	Yes
Command Reference	SC23-5846-01	Yes
Data Movement Utilities Guide and Reference	SC23-5847-01	Yes
Data Recovery and High Availability Guide and Reference	SC23-5848-01	Yes
Data Servers, Databases, and Database Objects Guide	SC23-5849-01	Yes
Database Security Guide	SC23-5850-01	Yes
Developing ADO.NET and OLE DB Applications	SC23-5851-01	Yes
Developing Embedded SQL Applications	SC23-5852-01	Yes
Developing Java Applications	SC23-5853-01	Yes
Developing Perl and PHP Applications	SC23-5854-01	No
Developing User-defined Routines (SQL and External)	SC23-5855-01	Yes
Getting Started with Database Application Development	GC23-5856-01	Yes
Getting Started with DB2 installation and administration on Linux and Windows	GC23-5857-01	Yes
Internationalization Guide	SC23-5858-01	Yes
Message Reference, Volume 1	GI11-7855-00	No
Message Reference, Volume 2	GI11-7856-00	No
Migration Guide	GC23-5859-01	Yes
Net Search Extender Administration and User's Guide	SC23-8509-01	Yes
Partitioning and Clustering Guide	SC23-5860-01	Yes
Query Patroller Administration and User's Guide	SC23-8507-00	Yes
Quick Beginnings for IBM Data Server Clients	GC23-5863-01	No

Table 47. DB2 technical information (continued)

Name	Form Number	Available in print
Quick Beginnings for DB2 Servers	GC23-5864-01	Yes
Spatial Extender and Geodetic Data Management Feature User's Guide and Reference	SC23-8508-01	Yes
SQL Reference, Volume 1	SC23-5861-01	Yes
SQL Reference, Volume 2	SC23-5862-01	Yes
System Monitor Guide and Reference	SC23-5865-01	Yes
Troubleshooting Guide	GI11-7857-01	No
Tuning Database Performance	SC23-5867-01	Yes
Visual Explain Tutorial	SC23-5868-00	No
What's New	SC23-5869-01	Yes
Workload Manager Guide and Reference	SC23-5870-01	Yes
pureXML Guide	SC23-5871-01	Yes
XQuery Reference	SC23-5872-01	No

Table 48. DB2 Connect-specific technical information

Name	Form Number	Available in print
Quick Beginnings for DB2 Connect Personal Edition	GC23-5839-01	Yes
Quick Beginnings for DB2 Connect Servers	GC23-5840-01	Yes
DB2 Connect User's Guide	SC23-5841-01	Yes

Table 49. Information Integration technical information

Name	Form Number	Available in print
Information Integration: Administration Guide for Federated Systems	SC19-1020-01	Yes
Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1018-02	Yes
Information Integration: Configuration Guide for Federated Data Sources	SC19-1034-01	No
Information Integration: SQL Replication Guide and Reference	SC19-1030-01	Yes
Information Integration: Introduction to Replication and Event Publishing	SC19-1028-01	Yes

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* DVD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at http://www.ibm.com/shop/publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
  1. Locate the contact information for your local representative from one of the following Web sites:
     - The IBM directory of world wide contacts at www.ibm.com/planetwide
     - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
  2. When you call, specify that you want to order a DB2 publication.
  3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see "DB2 technical library in hardcopy or PDF format" on page 355.

Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

For DB2 Version 9.5 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/

For DB2 Version 9 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

- To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the **Add...** button.

       Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
     - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.
- To display topics in your preferred language in a Firefox or Mozilla browser:
  1. Select the button in the **Languages** section of the **Tools** —> **Options** —> **Advanced** dialog. The Languages panel is displayed in the Preferences window.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
     - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

Updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. Non-Administrative and Non-Root DB2 Information Centers always run in stand-alone mode. .
2. Use the Update feature to see what updates are available. If there are updates that you would like to install, you can use the Update feature to obtain and install them

   Note: If your environment requires installing the DB2 Information Center updates on a machine that is not connected to the internet, you have to mirror the update site to a local file system using a machine that is connected to the internet and has the DB2 Information Center installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.
   If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.
3. Stop the stand-alone Information Center, and restart the DB2 Information Center on your computer.

Note: On Windows Vista, the commands listed below must be run as an administrator. To launch a command prompt or graphical tool with full administrator privileges, right-click on the shortcut and then select **Run as administrator**.

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv95 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the <Program Files>\IBM\DB2 Information Center\Version 9.5 directory, where <Program Files> represents the location of the Program Files directory.
     c. Navigate from the installation directory to the doc\bin directory.
     d. Run the help_start.bat file:
        ```
        help_start.bat
        ```
   - On Linux:

a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9.5 directory.
b. Navigate from the installation directory to the doc/bin directory.
c. Run the help_start script:

```
help_start
```

The systems default Web browser launches to display the stand-alone Information Center.

3. Click the **Update** button ( ). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the installation process, check the selections you want to install, then click **Install Updates**.
5. After the installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center:
   - On Windows, navigate to the installation directory's doc\bin directory, and run the help_end.bat file:

     ```
     help_end.bat
     ```

     Note: The help_end batch file contains the commands required to safely terminate the processes that were started with the help_start batch file. Do not use `Ctrl-C` or any other method to terminate help_start.bat.
   - On Linux, navigate to the installation directory's doc/bin directory, and run the help_end script:

     ```
     help_end
     ```

     Note: The help_end script contains the commands required to safely terminate the processes that were started with the help_start script. Do not use any other method to terminate the help_start script.
7. Restart the DB2 Information Center.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv95 start
     ```

The updated DB2 Information Center displays the new and updated topics.

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials

To view the tutorial, click on the title.

"pureXML™" in *pureXML Guide*
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

"Visual Explain" in *Visual Explain Tutorial*
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

DB2 documentation
Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

DB2 Technical Support Web site
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/support.html

Terms and Conditions

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

Personal use: You may reproduce these Publications for your personal, non commercial 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.

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.

Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This document may provide links or references to non-IBM Web sites and resources. IBM makes no representations, warranties, or other commitments whatsoever about any non-IBM Web sites or third-party resources that may be referenced, accessible from, or linked from this document. A link to a non-IBM Web site does not mean that IBM endorses the content or use of such Web site or

© IBM Corporation 1993, 2008

its owner. In addition, IBM is not a party to or responsible for any transactions you may enter into with third parties, even if you learn of such parties (or use a link to such parties) from an IBM site. Accordingly, you acknowledge and agree that IBM is not responsible for the availability of such external sites or resources, and is not responsible or liable for any content, services, products, or other materials on or available from those sites or resources. Any software provided by third parties is subject to the terms and conditions of the license that accompanies that software.

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

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

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

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

The licensed program described in this 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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which 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.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

The following terms are trademarks or registered trademarks of the International Business Machines Corporation in the United States, other countries, or both.

pureXML	OS/390
VisualAge	DB2 Universal Database
z/OS	Redbooks
System i	IBM
DB2	AIX
Lotus Notes	Rational
ibm.com	WebSphere

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds 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 are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Intel is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

.NET
  common language runtime
    external routine development support 145
    routine example 191
    routines 144, 146, 157, 159
  debugging CLR routines 162
  routines
    compile and link options 161

Numerics

32-bit support
  external routines 123
64-bit support
  external routines 123

A

ADMIN_CMD procedure 30
AIX
  C routines
    compile and link options 254
  C++ routines
    compile and link options 255
  C++ stored procedures
    building with configuration files 264
  C++ user-defined functions
    building with configuration files 265
  IBM COBOL routines
    building 279
    compile and link options 274
  Micro Focus COBOL routines
    compile and link options 275
ALLOCATE CURSOR statement
  caller routine 101
application development
  routines 1
ASSOCIATE RESULT SET LOCATOR statement 101
auditing
  transactions
    using SQL functions 32
authorizations
  for external routines 52, 334

B

backup
  external routine libraries 116
BASIC data types 323
BASIC language 321
BigDecimal data type
  Java 289
BIGINT data type
  COBOL 271
  Java 289
  OLE DB table function 330
  routines in Java (DB2GENERAL) 296
  user-defined functions (UDFs) 223
binary large objects (BLOBs)
  COBOL 271
  Java 289
  OLE DB table function 330
  routines
    Java (DB2GENERAL) 296
  User Defined Functions (UDFs)
    C/C++ 223
binding
  routines 52, 334
  SQL procedures 102
BLOB-FILE COBOL type 271
BLOB-LOCATOR COBOL type 271
BLOBs (binary large objects)
  COBOL 271
  Java 289
  OLE DB table function 330
  routines
    Java (DB2GENERAL) 296
  user-defined functions (UDFs
    C/C++ 223
books
  printed
    ordering 358
built-in routines 20

C

C language
  functions
    parameter styles 213
  procedures
    example 195
    parameter styles 210
    result sets 244
    XML support 195
    XQuery support 195
  routines 204
    32-bit routines on a 64-bit database server 124
    building 247, 253
    building on UNIX 248
    building on Windows 250
    compile options on AIX 254
    compile options on HP-UX 256
    compile options on Linux 259
    compile options on Solaris 261
    compile options on Windows 263
    creating 245
    dbinfo structure as parameter 215
    designing 206
    development support 205
    development tools 206
    include file 207
    null indicator parameters 209
    parameter passing 215
    parameter styles 209
    parameters 208
    PROGRAM TYPE clause 219
    result sets 215
    scratchpad as function parameter 218
    supported SQL data types in 220

© Copyright IBM Corp. 1993, 2008

C language *(continued)*
routines *(continued)*
syntax for passing arguments 230
C/C++ language
data types
OLE automation 323
functions
parameter styles 213
procedures
parameter styles 210
result sets 244
routines 204
32-bit routines on a 64-bit database server 124
building 247, 253
building on Windows 250
compile options on AIX 255
compile options on HP-UX 257
compile options on Linux 260
compile options on Solaris 262
compile options on Windows 263
creating 245
dbinfo structure as parameter 215
designing 206
development support 205
development tools 206
include file 207
null indicator parameters 209
parameter passing 215
parameter styles 209
parameters 208
PROGRAM TYPE clause 219
result sets 215
scratchpad as function parameter 218
supported SQL data types in 220
type decoration for routine bodies 242
C# .NET
routines
example 191
CALL procedures 341
from applications 342
from external routines 342
from SQL routines 343
from triggers 343
CALL statement
Command Line Processor 345
CASE statement 91
CAST FROM clause
data type handling 223
CHAR data type
COBOL 271
Java 289
OLE DB table function 330
routines, Java (DB2GENERAL) 296
user-defined functions (UDFs)
C/C++ 223
CHAR FOR BIT DATA data type 296
CLASSPATH environment variable 314
CLI (call level interface)
stored procedures
calling 347
CLOB-FILE COBOL type 271
CLOB-LOCATOR COBOL type 271
CLOBs (character large objects)
data type
COBOL 271
Java 289
OLE DB table function 330
CLOBs (character large objects) *(continued)*
data type *(continued)*
routines in Java (DB2GENERAL) 296
user-defined functions (UDFs) 223
CLP (command line processor)
terminating character 101
CLR (common language runtime)
procedures
returning result sets 151
routines 144
building 157, 159
compile and link options 161
creating 154, 155
design considerations 146
development support 145
development tools 146
examples of CLR procedures in C# 165
examples of CLR UDFs in C# 199
parameters 149
restrictions 153
security 152
XML support 191
XQuery support 191
CLR routines
.NET
debugging 162
COBOL data types
BLOB 271
BLOB-FILE 271
BLOB-LOCATOR 271
CLOB 271
CLOB-FILE 271
CLOB-LOCATOR 271
COMP-1 271
COMP-3 271
COMP-5 271
DBCLOB 271
DBCLOB-FILE 271
DBCLOB-LOCATOR 271
PICTURE (PIC) clause 271
USAGE clause 271
COBOL language
data types 271
IBM COBOL routines
building on AIX 279
building on Windows 281
compile options on AIX 274
compile options on Windows 277
Micro Focus routines
building on UNIX 280
building on Windows 282
compile options on AIX 275
compile options on HP-UX 276
compile options on Linux 277
compile options on Solaris 276
compile options on Windows 278
stored procedures 269
supported development software for external procedures 271
code pages
conversion routines 339
column types
creating
COBOL 271
COM.ibm.db2.app.Blob 296, 301
COM.ibm.db2.app.Clob 296, 301
COM.ibm.db2.app.Lob 301

COM.ibm.db2.app.StoredProc 298
COM.ibm.db2.app.UDF 294, 299
command line processor (CLP)
  creating routines 34
common language runtime
  function examples 176
  procedure examples 181
  procedures
    returning result sets 151
  routines 144
    building 157, 159
    creating 154, 155
    Dbinfo structure usage 149
    design considerations 146
    development support 145
    development tools 146
    errors 163
    examples 165, 176, 181
    examples of CLR functions in C# 199
    examples of CLR procedures in C# 165
    parameters 149
    restrictions 153
    scratchpad 149
    security 152
    supported SQL data types in 147
COMP-1 data types
  COBOL 271
COMP-3 data types
  COBOL 271
COMP-5 data types
  COBOL 271
comparison
  of functional types of routines 15
compound SQL
  SQL procedures 87
condition handlers
  CONTINUE clause 86
  SQL procedures
    description 99
configuration parameters
  javaheapsz configuration parameter 314
  jdk11path configuration parameter 314
contexts
  setting in multithreaded DB2 applications
    SQLJ routines 291
CREATE FUNCTION statement
  CAST FROM clause 223
  LANGUAGE OLE clause 321
  OLE automation routines 321
  PARAMETER STYLE clause 213, 293
  RETURNS clause 223
CREATE METHOD statement
  PARAMETER STYLE clause 293
CREATE PROCEDURE statement
  PARAMETER STYLE clause 210, 292
  PROGRAM TYPE clause 219
  with SQL procedures 101
CREATE ROUTINE statement
  PARAMETER STYLE clause 209
creating
  routines 6, 59, 66, 143, 245
    C/C++ 206
    common language runtime 154, 155
    Data Studio 35, 62
    development tools for SQL 62
cross-platform support
  Invoking 32-bit routines on a 64-bit database server 339
cursors
  SQL procedures 88
  XQuery 89

D

Data Studio
  overview 35
  SQL routine support 62
data types
  ARRAY 302
  COBOL 271
  conversion
    COBOL 271
    OLE automation types 322
  Java programs
    overview 289
DATE data type
  COBOL 271
  DB2GENERAL routines 296
  Java 289
  OLE DB table function 330
DB2 Information Center
  languages 359
  updating 360
  versions 359
  viewing in different languages 359
DB2_USE_DB2JCCT2_JROUTINE variable
  specifying drivers 287
DB2GENERAL parameter style 124
DB2GENERAL routines
  Java classes
    COM.ibm.db2.app.Blob 301
    COM.ibm.db2.app.Clob 301
    COM.ibm.db2.app.Lob 301
    COM.ibm.db2.app.StoredProc 298
    COM.ibm.db2.app.UDF 299
    overview 297
  overview 294
  stored procedures 298
  user-defined functions 294, 299
DB2SQL parameter style for external routines 124
DBCLOB data type
  COBOL 271
  Java 289
  OLE DB table function 330
  routines
    Java (DB2GENERAL) 296
  user-defined functions (UDFs)
    C/C++ 223
DBCLOB-FILE COBOL data type 271
DBCLOB-LOCATOR COBOL data type 271
dbinfo argument
  table functions 133
DBINFO option
  code pages 339
debugging
  routines
    .NET CLR 162
DECIMAL data type
  conversion
    COBOL 271
    Java 289
  DB2GENERAL routines 296
  OLE DB table function 330
  user-defined functions (UDFs) 223

DECLARE statements
  SQL procedures
    condition handlers 91
    conditions 91
    cursors 91
    variables 85, 91
distinct types
  passing to routines 351
documentation
  overview 355
  PDF 355
  printed 355
  terms and conditions of use 362
DOUBLE data type
  Java 289
  user-defined functions (UDFs)
    C/C++ 223
dynamic SQL
  SQL procedures comparison 69

E

errors
  .NET CLR routines 163
EXECUTE privilege
  routines 52, 334
external procedures
  COBOL 271
external routines
  32-bit support 123
  64-bit support 123
  APIs 21, 22, 117
  class files
    modifying 115
    restores 116
    security 114
  classes
    deploying 114
  creating 112, 143
  examples 316
  features 111
  libraries
    backing up 116
    deploying 114
    managing 116
    modifying 115
    performance 116
    restoring 116
    security 114
  naming conflicts 115
  overview 21, 111
  parameter styles 124
  performance 116
  programming languages 21, 22, 117
  SQL routines comparison 67

F

FLOAT data type
  COBOL 271
  Java 289
  Java routines (DB2GENERAL) 296
  OLE DB table function 330
  user-defined functions (UDFs)
    C/C++ 223
floating point
  parameter 223
FOR statement 93
functions
  comparison
    with other functional types of routines 15
  external
    features 131
  invoking 349
  Java
    PARAMETER STYLE clause 293
  overview 10
  parameters
    PARAMETER STYLE clause 213
  routine overview 30
  row functions 13
  scalar functions 12
  selection 350
  selection algorithm 350
  syntax references 349
  table functions 13

G

GENERAL parameter style for external routines 124
GENERAL WITH NULLS parameter style for external routines 124
GOTO statement 96
GRAPHIC data type
  COBOL 271
  Java 289
  OLE DB table function 330
  routines
    Java (DB2GENERAL) 296
graphic host variables
  routines 242
GRAPHIC parameter 223

H

help
  configuring language 359
  SQL statements 358
host variables
  COBOL data types 271
how this book is structured vii
HP-UX
  compile options
    C routines 256
    C++ routines 257
    Micro Focus COBOL routines 276
  link options
    C routines 256
    C++ routines 257
    Micro Focus COBOL routines 276

I

IBM Software Development Kit (SDK)
  developing external Java routines 285
IF statement 91, 92
infix notation
  user-defined functions (UDFs) 350
Int Java data type 289
INTEGER data type
  COBOL 271

INTEGER data type *(continued)*
Java 289
OLE DB table function 330
routines
Java (DB2GENERAL) 296
user-defined functions (UDFs)
C/C++ 223
invoking
routines 333
stored procedures 341
UDFs 352
user-defined table functions 353
ITERATE statement 97

J

Java
class files 314
classes 316
CLASSPATH environment variable 314
data types 289
functions 293
JAR files 315
methods
COM.ibm.db2.app.Blob 301
COM.ibm.db2.app.Clob 301
COM.ibm.db2.app.Lob 301
COM.ibm.db2.app.StoredProc 298
COM.ibm.db2.app.UDF 294, 299
PARAMETER STYLE JAVA 293
PARAMETER STYLE DB2GENERAL 294
PARAMETER STYLE JAVA 292, 293
procedures
PARAMETER STYLE JAVA 292
routines
building (JDBC) 311
building (overview) 311
building (SQLJ) 312
designing 288
development tools 288
drivers 287
overview 285
PARAMETER STYLE DB2GENERAL 294
parameter styles 124, 291
restrictions 306
supported development software 285
UNIX 286
stored procedures
JAR files 315
overview 285
table functions execution model 135, 307
UDFs (user-defined functions)
CALL statement for JAR files 315
DB2GENERAL 294
FENCED 294
NOT FENCED 294
scratchpads 294
Java Database Connectivity (JDBC)
routines
APIs 286
ARRAY data type 302, 317
building (overview) 311
building (procedure) 311
creating 309
development tools 288
drivers 286
examples (summary) 316
Java Database Connectivity (JDBC) *(continued)*
routines *(continued)*
examples (using array data type) 317
examples (XML and XQuery support) 317
stored procedures 303
XML
example 317
java.math.BigDecimal Java data type 289
jdk_path configuration parameter
application development 267
routines
building (UNIX) 286
running (UNIX) 286

K

keepfenced configuration parameter
updating 267

L

LANGUAGE OLE clause
CREATE FUNCTION statement 321
large objects (LOBs)
passing to routines 352
LEAVE statement 98
libraries
shared
rebuilding routine 266
Linux
C routines
compile and link options 259
C++ routines
compile and link options 260
Micro Focus COBOL routines
compile and link options 277
LOBs (large objects)
passing to routines 352
LONG VARCHAR data type
COBOL 271
Java 289
OLE DB table function 330
routines
Java (DB2GENERAL) 296
user-defined functions (UDFs)
C/C++ 223
LONG VARCHAR FOR BIT DATA data type
routines
Java (DB2GENERAL) 296
LONG VARGRAPHIC data type
COBOL 271
Java 289
OLE DB table function 330
parameter to UDF 223
routines
Java (DB2GENERAL) 296
LOOP statement
SQL procedures 94

M

methods 14
comparison
with other functional types of routines 15
distinct types as parameters 351

methods *(continued)*
external
features 131
Java
PARAMETER STYLE clause 293
MODIFIES SQL DATA clause
SQL access levels in SQL routines 40
multi-threaded applications
SQLJ routines 291

N

notices 365
NUMERIC parameter 223
NUMERIC SQL data type
COBOL 271
Java 289
OLE DB table function 330
routines
Java (DB2GENERAL) 296

O

object instance
OLE automation routines 322
Object Linking and Embedding (OLE) 321
OLE automation
BSTR data type 323
class identifier (CLSID) 321
controllers 321
methods 321
OLECHAR data type 323
programmatic identifier (progID) 321
routines
defining 321
designs 321
invoking methods 322
object instances 322
SCRATCHPAD option 322
servers 321
string data types 323
OLE DB
data types
converting to SQL data types 330
rowset names, fully qualified 329
table functions
connection string in EXTERNAL NAME clause 327
CONNECTSTRING option 327
creating 327
user-defined 327
using server name 327
OLE routines
syntax for passing arguments 230
ordering DB2 books 358
overloading
routine names 337

P

parameter markers
examples 79
overview 79
parameter styles
overview 124
PARAMETER STYLE DB2GENERAL 291
PARAMETER STYLE JAVA 291
parameters
C/C++ routines 208
SQL procedures 79
performance
external routines 116
routines
benefits 1
recommendations 43
SQL procedures 104
PICTURE (PIC) clause in COBOL types 271
portability
routines 42
precompiling
SQL procedures 102
problem determination
information available 362
tutorials 362
procedures
ADMIN_CMD
overview 30
C/C++ result sets 244
calling
from applications 342
from external routines 342
from SQL routines 343
from triggers 343
overview 341
commit effects on XML parameters and variables 88
common language runtime (CLR)
examples 165
functions comparison 15
Java
PARAMETER STYLE JAVA clause 292
methods comparison 15
overview 9
parameters
PARAMETER STYLE JAVA clause 292
PARAMETER STYLE SQL clause 210
references 340
result sets
.NET CLR (C# examples) 165
.NET CLR (procedure) 151
SQL routines 101
rollback effects on XML parameters and variables 88
SQL
array support 77
components 74
compound statements 87
condition handlers 99
conditional statements 91
control flow statements 91, 92
designing 74
features 73
looping statements 93
overview 73
parameters 79
structure 75
transfer of control statements 96
uses 73
variables 85, 87
XML
parameters 86
variables 86

R

REAL SQL data type
  COBOL 271
  conversion
    in C and C++ routines 223
    in Java (DB2GENERAL) routines 296
  Java 289
  OLE DB table function 330
receiving
  result sets
    from an SQL routine 101
    in JDBC applications and routines 304
    in SQLJ applications and routines 305
REPEAT statement 95
restoring
  external routine libraries 116
restrictions
  routines 127
result sets
  receiving from an SQL routine 101
  receiving in JDBC applications and routines 304
  receiving in SQLJ applications and routines 305
  returning
    .NET CLR procedures 151
    JDBC stored procedures 303
    SQL procedures 100
    SQLJ stored procedures 303
RETURN statement 98
returning result sets
  from JDBC stored procedures 303
  from SQLJ stored procedures 303
RETURNS clause
  CREATE FUNCTION statement 223
routines
  acceptable SQL statements 40
  altering 113
  benefits 1
  built-in 19, 20, 27
  C/C++
    32-bit routines on a 64-bit database server 124
    building 247, 253
    creating 245
    description 204
    designing 206
    development support 205
    development tools 206
    graphic host variables 242
    include file 207
    null indicator parameters 209
    parameter passing 215
    parameter styles 209
    parameters 208, 215
    pass by reference 215
    pass by value 215
    performance 124
    PROGRAM TYPE clause 219
    result sets 215, 244
    scratchpad as function parameter 218
    sqludf_scrat structure 218
    supported SQL data types in 220
    xml data type support 140
  classes 113
  CLR
    errors 163
  COBOL
    xml data type support 140
routines *(continued)*
  code pages
    conversion 339
  common language runtime
    building 157, 159
    creating 154
    description 144
    design considerations 146
    development support 145
    development tools 146
    errors 163
    examples 165
    examples of CLR functions (UDFs) 199
    examples of CLR procedures in C# 165
    examples of Visual Basic .NET CLR functions 176
    examples of Visual Basic .NET CLR procedures 181
    EXECUTION CONTROL clause 152
    restrictions 153
    returning result sets 151
    scratchpad usage 149
    security 152
    supported SQL data types in 147
    xml data type support 140
  comparison 7, 15
    of functional types 15
    SQL and external 67
    system-defined and user-defined 8
  creating
    Data Studio 35
    security 50
  database administration 30
  DB2GENERAL
    COM.ibm.db2.app.Blob 301
    COM.ibm.db2.app.Clob 301
    COM.ibm.db2.app.Lob 301
    description 294
    Java classes 297
  definers of 5
  defining scratchpad structure 140
  description 1
  development tools 34
  EXECUTE privilege 52, 334
  external 19, 27, 29
    32-bit and 64-bit support 123
    authorizations for 52, 334
    backup and restore of library and class files 116
    C/C++ 204, 206, 247, 253
    common language runtime 144, 154, 155, 157, 159
    comparison to SQL 67
    creating 112, 143
    deployment of libraries and classes 114
    description 21
    features 111, 131
    forbidden statements 127
    Java 311
    library management 116
    modifying library and class files 115
    naming conflicts 115
    overview 1, 111
    parameter styles 124
    performance 116
    restrictions 127, 131
    security 114
    SQL statement support 35
    supported APIs and programming languages 21, 22, 117
    updating Java routines 316

routines *(continued)*
external *(continued)*
xml data type support 140
forbidden statements 127
function path 337
functional types of 8
functions 10
row functions 13
scalar functions 12
table functions 13
when to use 18
graphic host variables 242
implementations 19
built-in 20
comparison 29
sourced 20
SQL 20
interoperability 42
invoking 333
32-bit routines on a 64-bit database server 339
from other routines 42
security 50
issuing CREATE statements 101
Java 286
creating 308, 309
JAR files 315
JDBC 311
overview 285
restrictions 306
xml data type support 140
libraries 113
methods 14, 141
when to use 18
name 337
nested 338
NOT FENCED
security 50, 51
OLE automation
defining 321
overloading 337
overview 1
passing distinct types to 351
passing LOBs to 352
performance 43
portability 42
portability between 32-bit and 64-bit platforms 140
procedures 9, 141
when to use 18
reading conflicts 55
rebuilding shared libraries 266
receiving result sets 101
recursive 338
restrictions 127
scalar UDFs
overview 131
security 50, 51
sourced 19, 27, 29
SQL 19, 20, 27, 29
comparison to external 67
creating 66
development tools 62
overview 61
performance 104
SQL statement support 35
syntax for passing arguments 230
system-defined 1, 3, 5, 7, 30
when to use 8
routines *(continued)*
types 1, 3, 18
comparison 29
comparison of functional types 15
functional 8
supported SQL statements 35
user-defined 1, 3, 5, 6, 7, 31, 141
creating 59
determining which implementation to use 29
when to use 8
uses of 30, 31
WCHARTYPE precompiler option 242
writing 141
conflicts 55
row functions 13
row sets
OLE DB
fully qualified names 329

S

scalar functions
description 131
processing model 133
routines 12
SCRATCHPAD option
OLE automation routines 322
preserving state 136
user-defined functions (UDFs) 136
scratchpads
32-bit and 64-bit platforms 140
for UDFs and methods 136
Java UDFs 294
SDK
Java
UNIX 286
security
routines 50, 51
SET statement
setting variables in SQL procedures 85
shared libraries
rebuilding routine 266
short data type
Java 289
SMALLINT data type
COBOL 271
Java 289
OLE DB table function 330
routines
Java (DB2GENERAL) 296
user-defined functions (UDFs)
C/C++ 223
Solaris operating systems
Micro Focus COBOL routines
compile and link options 276
routines
C compile and link options 261
C++ compile and link options 262
sourced
routines 20
sourced routines 20
SQL (Structured Query Language)
parameter style for external routines 124
routines
SQL access levels in SQL-bodied routines 40
SQL data types
COBOL 271

SQL data types *(continued)*
converting to OLE DB data types 330
Java 289
routines
Java (DB2GENERAL) 296
supported in OLE automation 322
user-defined functions (UDFs)
C/C++ 223
SQL functions
comparison to SQL procedures 68
SQL statement support 35
table functions
that modify SQL data 32
SQL methods
SQL statement support 35
SQL procedural language
comparison of SQL PL and inline SQL PL 63
control flow statements 91
executing inline SQL PL 65
inline SQL PL
executing 65
overview 62
performance 104
SQL procedures 64
SQL procedures
ATOMIC compound statements 75
CALL statement 345
comparison to dynamic compound SQL 69
comparison to SQL functions 68
components 74
condition handlers 99
conditional statements 91
control flow statements 91, 92
creating 101
cursors 88
designing 74
features 73
labels 75
looping statements 93
FOR statement 93
LOOP statement 94
REPEAT statement 95
WHILE statement 95
NOT ATOMIC compound statements 75
overview 73
parameters 79
performance 104
precompile and bind options 102
returning result sets 100
rewriting as SQL UDFs 70
SQL statement support 35
SQLCODE and SQLSTATE variables 86
structure 75
transfer of control statements 96
GOTO statement 96
ITERATE statement 97
LEAVE statement 98
RETURN statement 98
uses 73
variables 85
SQL routines 20
comparison to external routines 67
creating 66
development tools 62
Data Studio 62
overview 61
SQL statements
allowed in routines 35, 40
displaying help 358
looping statements 93
related to variables 91
transfer of control statements 96
SQL-result argument
table functions 133
SQL-result-ind argument
table functions 133
SQLCODE
variables in SQL procedures 86
sqldbchar data type
in C/C++ routines 223
SQLJ
building routines 286, 312
routines
compile options on UNIX 313
compile options on Windows 313
connection contexts 291
creating 308
development tools 288
stored procedures
returning result sets 303
SQLSTATE
variables in SQL procedures 86
SQLUDF include file
C/C++ routines 207
statements
CREATE FUNCTION 59, 321
CREATE METHOD 59
CREATE PROCEDURE 59
stored procedures 9
AIX C++ configuration files 264
CALL statement 345
calling
CLI applications 347
COBOL 269
invoking 341
references (syntax of call references) 340
selection 340
selection algorithm 340
String Java data type 289
structured types
attributes
accessing attributes with methods 14
methods 14
system-defined
routines 5

T

table functions 13
Java execution model 135, 307
user-defined table functions 133
table user-defined functions (UDFs)
processing model 134
tables
access
routines reading and writing conflicts 55
table functions 13
terms and conditions
use of publications 362
TIME data types
COBOL 271
Java 289
Java routines 296

TIME data types *(continued)*
OLE DB table function 330
TIME parameter 223
TIMESTAMP data type
COBOL 271
Java 289
OLE DB table function 330
routines
Java (DB2GENERAL) 296
TIMESTAMP parameter 223
tools
routine development 34
transact SQL 9
troubleshooting
online information 362
tutorials 362
tutorials
problem determination 362
troubleshooting 362
Visual Explain 361
type decoration
C++ routine bodies 242
type mapping
OLE automation
BASIC types 323

U

UNIX
C routines
building 248
Micro Focus COBOL routines
building 280
SQLJ routines
compile options 313
updates
DB2 Information Center 360
USAGE clause in COBOL types 271
user-defined functions
AIX C++ configuration files 265
C/C++
arguments 223
BIGINT data type 223
BLOB data type 223
CHAR data type 223
CLOB data type 223
DBCLOB data type 223
DOUBLE data type 223
FLOAT data type 223
INTEGER data type 223
LONG VARCHAR data type 223
parameters 223
REAL data type 223
SMALLINT data type 223
VARCHAR FOR BIT DATA data type 223
VARGRAPHIC data type 223
common language runtime UDFs
examples in C# 199
date parameters 223
DETERMINISTIC 136
distinct types as parameters 351
FOR BIT DATA modifier 223
infix notation 350
invoking 352
Java
I/O restrictions 294
LOB values as parameters 352
user-defined functions *(continued)*
NOT DETERMINISTIC 136
OLE DB table functions 327
re-entrant 136
returning data 223
rewriting SQL procedures as UDFs 70
saving state 136
scalar
FINAL CALL 133
SCRATCHPAD option 136
scratchpad portability between 32-bit and 64-bit platforms 140
table 133
FINAL CALL 134
invoking 353
NO FINAL CALL 134
processing model 134
SQL-result argument 133
SQL-result-ind argument 133
uses of 31
user-defined routines 6

V

VARCHAR data type
COBOL 271
Java 289
Java (DB2GENERAL) routines 296
OLE DB table function 330
VARCHAR FOR BIT DATA data type
C/C++ user-defined functions 223
Java (DB2GENERAL) routines 296
VARGRAPHIC data type
C/C++ user-defined functions 223
COBOL 271
Java 289
Java (DB2GENERAL) routines 296
OLE DB table function 330
variables
SQL procedures 85, 91
Visual Explain
tutorial 361

W

wchar_t data type
C/C++ routines 223
WCHARTYPE NOCONVERT precompiler option 242
WHILE statement
SQL procedures 95
who should use this book vii
Windows operating systems
C/C++ routines
building 250
compile options 263
link options 263
COBOL routines
building 281
compile options 277
link options 277
Micro Focus COBOL routines
building 282
compile options 278
link options 278
SQLJ routines 313

X

XML
  data type 140
  parameters
    commits and rollbacks 88
    procedures 86
  variables in procedures 86
XQuery statements
  calling from SQL 89

Printed in USA

SC23-5855-01

Spine information:

DB2 Version 9.5 for Linux, UNIX, and Windows

Developing User-defined Routines (SQL and External)