IBM® DB2 Universal Database™

Application Development Guide: Programming Client Applications

Version 8.2

SC09-4826-01

IBM® DB2 Universal Database™

Application Development Guide: Programming Client Applications

Version 8.2

SC09-4826-01

Before using this information and the product it supports, be sure to read the general information under *Notices*.

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.

Contents

About this book

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

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

  - The simulation of EBCDIC binary collation.
- *Application Development Guide: Programming Server Applications* contains what you need to know about programming using server-side objects, including routines, large objects, user-defined types, and triggers. It includes information on:
  - Routines (stored procedures, user-defined functions, and methods), including:
    - Routine performance, security, library management considerations, and restrictions.
    - Creating routines, including external routines, and the CREATE statement.
    - Procedure parameter modes and parameter handling.
    - Procedure result sets.
    - SQL procedures including debugging and condition handling.
    - User-defined scalar and table functions.
    - User-defined scalar and table function calls (FIRST call, FINAL call,...) and scratchpads.
    - Methods.
    - Authorizations and binding of external routines.
    - Language-specific considerations for C, Java, .NET common language runtime, and OLE automation routines.
    - Invoking routines.
    - Function selection.
    - Passing distinct types and LOBs to functions.
    - Code pages and routines.
  - Large objects, including LOB usage and locators, reference variables, and CLOB data.
  - User-defined distinct types, including strong typing, defining and dropping UDTs, creating tables with structured types, using distinct types and typed tables for specific applications, manipulating distinct types and casting between them, and performing comparisons and assignments with distinct types, including UNION operations on distinctly typed columns.
  - User-defined structured types, including storing instances and instantiation, structured type hierarchies, defining structured type behavior, the dynamic dispatch of methods, the comparison, casting, and constructor functions, and mutator and observer methods for structured types.
  - Typed tables, including creating, dropping, substituting, storing objects, defining system-generated object identifiers, and constraints on object identifier columns.
  - Reference types, including relationships between objects in typed tables, semantic relationships with references, and referential integrity versus scoped references.
  - Typed tables and typed views, including structured types as column types, transform functions and transform groups, host language program mappings, and structured type host variables.
  - Triggers, including INSERT, UPDATE, and DELETE triggers, interactions with referential constraints, creation guidelines, granularity, activation time, transition variables and tables, triggered actions, multiple triggers, and synergy between triggers, constraints, and routines.
- *Application Development Guide: Building and Running Applications* contains what you need to know to build and run DB2 applications on the operating systems supported by DB2:
  - AIX

- HP-UX
- Linux
- Solaris
- Windows

It includes information on:

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

Part 1. Introduction

Chapter 1. Overview of Supported Programming Interfaces

DB2 Universal Database tools for developing applications

You can use a variety of different tools when developing your applications. DB2® Universal Database supplies the following tools to help you write and test the SQL statements in your applications, and to help you monitor their performance.

Note: Not all tools are available on every platform.

Control Center

A graphical interface that displays database objects (such as databases, tables, and packages) and their relationship to each other. Use the Control Center to perform administrative tasks such as configuring the system, managing directories, backing up and recovering the system, scheduling jobs, and managing media.

DB2 also provides the following facilities:

Command Editor
: Is used to enter DB2 commands and SQL statements in an interactive window, and to see the execution result in a result window. You can scroll through the results and save the output to a file.

Task Center
: Is used to create scripts, which you can store and invoke at a later time. These scripts can contain DB2 commands, SQL statements, or operating system commands. You can schedule scripts to run unattended. You can run these jobs once or you can set them up to run on a repeating schedule. A repeating schedule is particularly useful for tasks like backups. The Task Center can also be used to monitor your system for early warnings of potential problems, or to automate actions to correct problems.

Journal
: Is used to view the following types of information: all available information about jobs that are pending execution, executing, or that have completed execution; the recovery history log; the alerts log; and the messages log. You can also use the Journal to review the results of jobs that run unattended.

Tools Setting
Is used to change the settings for the Task Center.

Event Monitor
Collects performance information on database activities over a period of time. Its collected information provides a good summary of the activity for a particular database event: for example, a database connection or an SQL statement.

Visual Explain

An installable option for the Control Center, Visual Explain is a graphical interface that enables you to analyze and tune SQL statements, including viewing access plans chosen by the optimizer for SQL statements.

IBM DB2 Development Add-In overview

The IBM® DB2® Development Add-In is a collection of features that integrates seamlessly into your Microsoft® Visual Studio .NET development environment for working with DB2 servers and developing DB2 routines. With the add-in, you can:

- Launch various DB2 development and administration tools
- Create and manage DB2 projects in the Solution Explorer
- Access and manage DB2 data connections in the IBM Explorer
- Create and modify DB2 scripts, including scripts to create stored procedures and user-defined functions (UDFs)

DB2 Tools toolbar:

The DB2 Tools toolbar enables you to launch the various DB2 development and administration tools. With the DB2 Tools toolbar, you can launch the following DB2 tools:

- Development Center
- Control Center
- Replication Center
- Command Editor
- Task Center
- Health Center
- Journal

DB2 Project type:

The IBM DB2 Development Add-In introduces a new IBM Projects folder, which includes a DB2 Database Project type for developing DB2 database server scripts. With a DB2 Project, you can:

- Add new or existing SQL stored procedure scripts
- Add new or existing SQL UDF scripts
- Add new or existing scripts based on generic templates
- Specify build configuration options including script build order
- Check your script files into Microsoft Visual Source Safe source control management system

Data Connections folder in the IBM Explorer:

The IBM DB2 Development Add-In extends the Visual Studio .NET environment by adding a new tool called IBM Explorer, a dockable window that is similar to the Visual Studio .NET Server Explorer. The IBM Explorer provides Visual Studio .NET users with access to IBM database connections using the Data Connections folder. The Data Connections folder in the IBM Explorer is specifically designed for DB2 managed provider connections. From the Data Connections folder in the IBM Explorer, you can:

- Work with multiple named DB2 connections supporting connect of demand technology
- Specify database catalog filters and local caching for higher performance and scalability
- View properties of server objects including tables, views, stored procedures, and functions
- Retrieve data from tables and views
- Execute test runs for stored procedures and UDFs
- View source code for stored procedures and functions
- Generate ADO .NET code using drag and drop

DB2 SQL Editor:

The IBM DB2 Development Add-In also provides you with a DB2 SQL Editor. With the editor, you can change and view the code in your DB2 routines and scripts. The DB2 SQL Editor includes the following features:

- Colorized text for increased readability of the SQL.
- Integration with the Microsoft Visual Studio .NET IntelliSense feature, which allows for intelligent auto-completion while typing DB2 scripts.

Supported Programming Interfaces

The sections that follow provide an overview of the supported programming interfaces.

DB2 Supported Programming Interfaces

You can use several different programming interfaces to manage or access DB2® databases. You can:

- Use DB2 APIs to perform administrative functions such as backing up and restoring databases.
- Embed static and dynamic SQL statements in your applications.
- Code DB2 Call Level Interface (DB2 CLI) function calls in your applications to invoke dynamic SQL statements.
- Develop Java™ applications and applets that call the Java Database Connectivity application programming interface (JDBC API).
- Develop Microsoft® Visual Basic and Visual C++ applications that conform to Data Access Object (DAO) and Remote Data Object (RDO) specifications, and ActiveX Data Object (ADO) applications that use the OLE DB Bridge.
- Develop ADO.NET applications using DB2 .NET Data Provider, OLE DB .NET Data Provider or ODBC .NET Data Provider.
- Develop applications using IBM® or third-party tools such as Net.Data®, Excel, Perl, and Open Database Connectivity (ODBC) end-user tools such as Lotus® Approach, and its programming language, LotusScript.

The way your application accesses DB2 databases will depend on the type of application you want to develop. For example, if you want a data entry application, you might choose to embed static SQL statements in your application. If you want an application that performs queries over the World Wide Web, you might choose Net.Data, Perl, or Java.

Apart from how the application accesses data, you also need to consider the following:

- Controlling data values using:
  - Data types (built-in or user-defined)
  - Table check constraints
  - Referential integrity constraints
  - Views using the CHECK OPTION
  - Application logic and variable types
- Controlling the relationship between data values using:
  - Referential integrity constraints
  - Triggers
  - Application logic
- Executing programs at the server using:
  - Stored procedures
  - User-defined functions
  - Triggers

You will notice that this list mentions some capabilities more than once, such as triggers. This reflects the flexibility of these capabilities to address more than one design criteria.

Your first and most fundamental decision is whether or not to move the logic to enforce application related rules about the data into the database.

The key advantage in transferring logic focused on the data from the application into the database is that your application becomes more independent of the data. The logic surrounding your data is centralized in one place, the database. This means that you can change data or data logic once and affect *all* applications immediately.

This latter advantage is very powerful, but you must also consider that any data logic put into the database affects *all* users of the data equally. You must consider whether the rules and constraints that you wish to impose on the data apply to all users of the data or just the users of your application.

Your application requirements may also affect whether to enforce rules at the database or the application. For example, you may need to process validation errors on data entry in a specific order. In general, you should do these types of data validation in the application code.

You should also consider the computing environment where the application is used. You need to consider the difference between performing logic on the client machines against running the logic on the usually more powerful database server machines using either stored procedures, UDFs, or a combination of both.

In some cases, the correct answer is to include the enforcement in both the application (perhaps due to application specific requirements) and in the database (perhaps due to other interactive uses outside the application).

Related concepts:

- "DB2 Call Level Interface (CLI) versus embedded dynamic SQL" on page 126
- "Embedded SQL" on page 7
- "DB2 Call Level Interface" on page 9
- "DB2 Application Programming Interfaces" on page 7
- "ActiveX Data Objects and Remote Data Objects" on page 12
- "Perl DBI" on page 13
- "ODBC End-User Tools" on page 14
- "Tools for Building Web Applications" on page 14
- "Java Database Connectivity (JDBC)" on page 11

DB2 Application Programming Interfaces

Your applications may need to perform some database administration tasks, such as creating, activating, backing up, or restoring a database. DB2® provides numerous APIs so you can perform these tasks from your applications, including embedded SQL and DB2 CLI applications. This enables you to program the same administrative functions into your applications that you can perform using the DB2 server administration tools available in the Control Center.

Additionally, you might need to perform specific tasks that can only be performed using the DB2 APIs. For example, you might want to retrieve the text of an error message so your application can display it to the end user. To retrieve the message, you must use the Get Error Message API.

Related concepts:

- "Authorization Considerations for APIs" on page 48
- "Administrative APIs in Embedded SQL or DB2 CLI Programs" on page 40

Embedded SQL

Structured Query Language (SQL) is the database interface language used to access and manipulate data in DB2® databases. You can embed SQL statements in your applications, enabling them to perform any task supported by SQL, such as retrieving or storing data. Using DB2, you can code your embedded SQL applications in the C/C++, COBOL, FORTRAN, Java™ (SQLJ), and REXX programming languages.

Note: The REXX and Fortran programming languages have not been enhanced since Version 5 of DB2 Universal Database.

An application in which you embed SQL statements is called a host program. The programming language you use to create a host program is called a host language. The program and language are defined this way because they host or accommodate SQL statements.

For static SQL statements, you know before compile time the SQL statement type and the table and column names. The only unknowns are specific data values the statement is searching for or updating. You can represent those values in host

language variables. You precompile, bind and then compile static SQL statements before you run your application. Static SQL is best run on databases whose statistics do not change a great deal. Otherwise, the statements will soon get out of date.

In contrast, dynamic SQL statements are those that your application builds and executes at run time. An interactive application that prompts the end user for key parts of an SQL statement, such as the names of the tables and columns to be searched, is a good example of dynamic SQL. The application builds the SQL statement while it's running, and then submits the statement for processing.

You can write applications that have static SQL statements, dynamic SQL statements, or a mix of both.

Generally, static SQL statements are well-suited for high-performance applications with predefined transactions. A reservation system is a good example of such an application.

Generally, dynamic SQL statements are well-suited for applications that run against a rapidly changing database where transactions need to be specified at run time. An interactive query interface is a good example of such an application.

When you embed SQL statements in your application, you must precompile and bind your application to a database with the following steps:

1. Create source files that contain programs with embedded SQL statements.
2. Connect to a database, then precompile each source file.

   The precompiler converts the SQL statements in each source file into DB2 run-time API calls to the database manager. The precompiler also produces an access package in the database and, optionally, a bind file, if you specify that you want one created.

   The access package contains access plans selected by the DB2 optimizer for the static SQL statements in your application. The access plans contain the information required by the database manager to execute the static SQL statements in the most efficient manner as determined by the optimizer. For dynamic SQL statements, the optimizer creates access plans when you run your application.

   The bind file contains the SQL statements and other data required to create an access package. You can use the bind file to re-bind your application later without having to precompile it first. The re-binding creates access plans that are optimized for current database conditions. You need to re-bind your application if it will access a different database from the one against which it was precompiled. You should re-bind your application if the database statistics have changed since the last binding.
3. Compile the modified source files (and other files without SQL statements) using the host language compiler.
4. Link the object files with the DB2 and host language libraries to produce an executable program.
5. Bind the bind file to create the access package if this was not already done at precompile time, or if a different database is going to be accessed.
6. Run the application. The application accesses the database using the access plan in the package.

Related concepts:

- "Embedded SQL in REXX Applications" on page 495
- "Embedded SQL Statements in C and C++" on page 135
- "Embedded SQL Statements in COBOL" on page 178
- "Embedded SQL Statements in FORTRAN" on page 199
- "Embedded SQL for Java (SQLJ)" on page 12

Related tasks:

- "Embedding SQL Statements in a Host Language" on page 55

DB2 Call Level Interface

DB2® CLI is a programming interface that your C and C++ applications can use to access DB2 databases. DB2 CLI is based on the Microsoft® Open Database Connectivity (ODBC) specification, and the ISO CLI standard. Since DB2 CLI is based on industry standards, application programmers who are already familiar with these database interfaces may benefit from a shorter learning curve.

When you use DB2 CLI, your application passes dynamic SQL statements as function arguments to the database manager for processing. As such, DB2 CLI is an alternative to embedded dynamic SQL.

It is also possible to run the SQL statements as static SQL in a CLI, ODBC or JDBC application. The CLI/ODBC/JDBC Static Profiling feature enables end users of an application to replace the use of dynamic SQL with static SQL in many cases.

You can build an ODBC application without using an ODBC driver manager, and simply use DB2's ODBC driver on any platform by linking your application with `libdb2` on UNIX®, and `db2cli.lib` on Windows® operating systems. The DB2 CLI sample programs demonstrate this. They are located in `sqllib/samples/cli` on UNIX and `sqllib\samples\cli` on Windows operating systems.

You do not need to precompile or bind DB2 CLI applications because they use common access packages provided with DB2. You simply compile and link your application.

However, before your DB2 CLI or ODBC applications can access DB2 databases, the DB2 CLI bind files that come with the DB2 AD Client must be bound to each DB2 database that will be accessed. This occurs automatically with the execution of the first statement, but we recommend that the database administrator bind the bind files from one client on each platform that will access a DB2 database.

For example, suppose you have AIX®, Solaris Operating Environment, and Windows 98 clients that each access two DB2 databases. The administrator should bind the bind files from one AIX client on each database that will be accessed. Next, the administrator should bind the bind files from one Solaris Operating Environment client on each database that will be accessed. Finally, the administrator should do the same on one Windows 98 client.

Related concepts:

- "Administrative APIs in Embedded SQL or DB2 CLI Programs" on page 40
- "DB2 CLI versus Embedded Dynamic SQL" on page 10

Related tasks:

- "Creating static SQL with CLI/ODBC/JDBC Static Profiling" in the *CLI Guide and Reference, Volume 1*

DB2 CLI versus Embedded Dynamic SQL

You can develop dynamic applications using either embedded dynamic SQL statements or DB2® CLI. In both cases, SQL statements are prepared and processed at run time. Each method has unique advantages.

The advantages of DB2 CLI are as follows:

Portability DB2 CLI applications use a standard set of functions to pass SQL statements to the database. All you need to do is compile and link DB2 CLI applications before you can run them. In contrast, you must precompile embedded SQL applications, compile them, and then bind them to the database before you can run them. This process effectively ties your application to a particular database.

No binding You do not need to bind individual DB2 CLI applications to each database they access. You only need to bind the bind files that are shipped with DB2 CLI once for all your DB2 CLI applications. This can significantly reduce the amount of time you spend managing your applications.

Extended fetching and input
DB2 CLI functions enable you to retrieve multiple rows in the database into an array with a single call. They also let you execute an SQL statement many times using an array of input variables.

Consistent interface to catalog
Database systems contain catalog tables that have information about the database and its users. The form of these catalogs can vary among systems. DB2 CLI provides a consistent interface to query catalog information about components such as tables, columns, foreign and primary keys, and user privileges. This shields your application from catalog changes across releases of database servers, and from differences among database servers. You don't have to write catalog queries that are specific to a particular server or product version.

Extended data conversion
DB2 CLI automatically converts data between SQL and C data types. For example, fetching any SQL data type into a C char data type converts it into a character-string representation. This makes DB2 CLI well-suited for interactive query applications.

No global data areas
DB2 CLI eliminates the need for application controlled, often complex global data areas, such as SQLDA and SQLCA, typically associated with embedded SQL applications. Instead, DB2 CLI automatically allocates and controls the necessary data structures, and provides a handle for your application to reference them.

Retrieve result sets from stored procedures
DB2 CLI applications can retrieve multiple rows and result sets generated from a stored procedure residing on a DB2 Universal Database™ server, a DB2 for MVS™/ESA server (Version 5 or later), or an OS/400® server (Version 5 or later). Support for multiple result sets retrieval on OS/400 requires that PTF (Program

Temporary Fix) SI01761 be applied to the server. Contact your OS/400 system administrator to ensure that this PTF has been applied.

Scrollable cursors
DB2 CLI supports server-side scrollable cursors that can be used in conjunction with array output. This is useful in GUI applications that display database information in scroll boxes that make use of the Page Up, Page Down, Home and End keys. You can declare a cursor as scrollable and then move forwards or backwards through the result set by one or more rows. You can also fetch rows by specifying an offset from the current row, the beginning or end of a result set, or a specific row you bookmarked previously.

The advantages of embedded dynamic SQL are as follows:

Granular Security
All DB2 CLI users share the same privileges. Embedded SQL offers the advantage of more granular security through granting execute privileges to particular users for a package.

More Supported Languages
Embedded SQL supports more than just C and C++. This might be an advantage if you prefer to code your applications in another language.

More Consistent with Static SQL
Dynamic SQL is generally more consistent with static SQL. If you already know how to program static SQL, moving to dynamic SQL might not be as difficult as moving to DB2 CLI.

Related concepts:

- "DB2 Call Level Interface (CLI) versus embedded dynamic SQL" on page 126
- "Advantages of DB2 CLI over embedded SQL" on page 127
- "When to use DB2 CLI or embedded SQL" on page 129

Java Database Connectivity (JDBC)

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

JDBC can be especially useful for accessing DB2 databases across the Internet. Using the Java programming language, you can develop JDBC applets and applications that access and manipulate data in remote DB2 databases using a network connection. You can also create JDBC stored procedures that reside on the server, access the database server, and return information to a remote client application that calls the stored procedure.

The JDBC API, which is similar to the CLI/ODBC API, provides a standard way to access databases from Java code. Your Java code passes SQL statements as method arguments to the DB2 JDBC driver. The driver handles the JDBC API calls from your client Java code.

Java's portability enables you to deliver DB2 access to clients on multiple platforms, requiring only a Java-enabled web browser, or a Java runtime environment.

JDBC Type 2

Java applications based on the JDBC type 2 driver rely on the DB2 client to connect to DB2. You start your application from the desktop or command line, like any other application. The DB2 JDBC driver handles the JDBC API calls from your application, and uses the client connection to communicate the requests to the server and to receive the results. You cannot create Java applets using the JDBC type 2 driver.

Note: The JDBC type 2 driver is recommended for WebSphere® Application Servers.

JDBC Type 3

If you use the JDBC type 3 driver, you can only create Java applets. Java applets do not require the DB2 client to be installed on the client machine. Typically, you would embed the applet in a HyperText Markup Language (HTML) web page.

To run an applet based on the JDBC type 3 driver, you need only a Java-enabled web browser or applet viewer on the client machine. When you load your HTML page, the browser downloads the Java applet to your machine, which then downloads the Java class files and DB2's JDBC driver. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes a separate network connection with the DB2 database through the JDBC applet server residing on the Web server.

Note: The JDBC type 3 driver is deprecated for Version 8.

JDBC Type 4

You can use the JDBC type 4 driver, which is new for Version 8, to create both Java applications and applets. To run an application or an applet that is based on the type 4 driver, you only require the `db2jcc.jar` file. No DB2 client is required.

For more information on DB2 JDBC support, visit the Web page at:

`http://www.ibm.com/software/data/db2/udb/ad/v8/java`

Embedded SQL for Java (SQLJ)

DB2® Java™ embedded SQL (SQLJ) support is provided by the DB2 AD Client. With DB2 SQLJ support, in addition to DB2 JDBC support, you can build and run SQLJ applets, applications, and stored procedures. These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

For more information on DB2 SQLJ support, visit the Web page at:

`http://www.ibm.com/software/data/db2/udb/ad/v8/java`

ActiveX Data Objects and Remote Data Objects

You can write Microsoft® Visual Basic and Microsoft Visual C++ database applications that conform to the Data Access Object (DAO) and Remote Data

Object (RDO) specifications. DB2® also supports ActiveX Data Object (ADO) applications that use the Microsoft OLE DB to ODBC Bridge.

ActiveX Data Objects (ADO) allow you to write an application to access and manipulate data in a database server through an OLE DB provider. The primary benefits of ADO are high speed development time, ease of use, and a small disk footprint.

Remote Data Objects (RDO) provide an information model for accessing remote data sources through ODBC. RDO offers a set of objects that make it easy to connect to a database, execute queries and stored procedures, manipulate results, and commit changes to the server. It is specifically designed to access remote ODBC relational data sources, and makes it easier to use ODBC without complex application code.

For full samples of DB2 applications that use the ADO and RDO specifications, see the following directories:
- For Visual Basic ActiveX Data Object samples, refer to `sqllib\samples\VB\ADO`
- For Visual Basic Remote Data Object samples, refer to `sqllib\samples\VB\RDO`
- For Visual Basic Microsoft Transaction Server samples, refer to sqllib\samples\VB\MTS
- For Visual C++ ActiveX Data Object samples, refer to `sqllib\samples\VC\ADO`

Related tasks:
- "Building ADO applications with Visual Basic" in the *Application Development Guide: Building and Running Applications*
- "Building RDO applications with Visual Basic" in the *Application Development Guide: Building and Running Applications*
- "Building ADO applications with Visual C++" in the *Application Development Guide: Building and Running Applications*

Related reference:
- "Visual Basic samples" in the *Application Development Guide: Building and Running Applications*
- "Visual C++ samples" in the *Application Development Guide: Building and Running Applications*

Perl DBI

DB2® supports the Perl Database Interface (DBI) specification for data access through the DBD::DB2 driver. The DB2 Universal Database™ Perl DBI website is located at:

```
http://www.ibm.com/software/data/db2/perl/
```

and contains the latest DBD::DB2 driver, and related information.

Perl is an interpreted language and the Perl DBI Module uses dynamic SQL. This makes Perl an ideal language for quickly creating and revising prototypes of DB2 applications. The Perl DBI Module uses an interface that is quite similar to the CLI and JDBC interfaces. This makes it easy to port Perl prototypes to CLI and JDBC.

Related concepts:
- "Programming Considerations for Perl" on page 489

ODBC End-User Tools

You can use ODBC end-user tools such as Lotus® Approach, Microsoft® Access, and Microsoft Visual Basic to create applications. ODBC tools provide a simpler alternative to developing applications than using a high-level programming language.

Lotus Approach provides two ways to access DB2® data. You can use the graphical interface to perform queries, develop reports, and analyze data. Or you can develop applications using LotusScript, a full-featured, object-oriented programming language that comes with a wide array of objects, events, methods, and properties, along with a built-in program editor.

DB2 .NET Data Provider

The DB2® .NET Data Provider extends DB2 support for the ADO.NET interface. The DB2 .NET Data Provider delivers high-performing, secure access to DB2 data.

The DB2 .NET Data Provider allows your .NET applications to access the following database management systems:

- DB2 Universal Database™ Version 8 for Windows®, UNIX®, and Linux-based computers
- DB2 Universal Database Version 6 (or later) for OS/390® and z/OS™, through DB2 Connect™
- DB2 Universal Database Version 5, Release 1 (or later) for AS/400® and iSeries™, through DB2 Connect
- DB2 Universal Database Version 7.3 (or later) for VSE & VM, through DB2 Connect

To develop and run applications that use DB2 .NET Data Provider you need the .NET Framework, Version 1.0 or 1.1.

In addition to the DB2 .NET Data Provider, there is also a collection of add-ins to the Microsoft® Visual Studio .NET IDE. These add-ins simplify the creation of DB2 applications that use the ADO.NET interface. You can also use these add-ins to develop server-side objects, such as SQL stored procedures and user-defined functions.

Sample applications in VB.NET and C#.NET demonstrating the DB2 .NET Data Provider are available at:

```
http://www.ibm.com/software/data/db2/udb/ad/v8/samples.html
```

Web Applications

The sections that follow describe the products and functions that are available for building Web applications.

Tools for Building Web Applications

DB2® Universal Database supports all the key Internet standards, making it an ideal database for use on the Web. It has in-memory speed to facilitate Internet searches and complex text matching combined with the scalability and availability

characteristics of a relational database. Because DB2 Universal Database supports WebSphere®, Java™ and XML Extender, it makes it easy for you to deploy your e-business applications.

DB2 Universal Developer's Edition has several tools that provide Web enablement support. WebSphere Studio Application Developer, Version 4, is an integrated development environment (IDE) that enables you to build, test, and deploy Java applications to a WebSphere Application Server and DB2 Universal Database. WebSphere Studio is a suite of tools that brings all aspects of Web site development into a common interface. WebSphere Application Server Advanced Edition (single-server) provides a robust deployment environment for e-business applications. Its components let you build and deploy personalized, dynamic Web content quickly and easily.

Related concepts:

- "WebSphere Studio" on page 15
- "XML Extender" on page 16

WebSphere Studio

WebSphere® Studio is a suite of tools that brings all aspects of Web site development into a common interface. The WebSphere Studio makes it easier than ever to cooperatively create, assemble, publish, and maintain dynamic interactive Web applications. The Studio is composed of the Workbench, the Page Designer, the Remote Debugger, and wizards, and it comes with trial copies of companion Web development products, such as Macromedia Flash, Fireworks, Freehand, and Director. WebSphere Studio enables you to do everything you need to create interactive Web sites that support your advanced business functions.

WebSphere Application Server Standard Edition (provided with DB2® Universal Developer's Edition) is a component of WebSphere Studio. It combines the portability of server-side business applications with the performance and manageability of Java™ technologies to offer a comprehensive platform for designing Java-based Web applications. It enables powerful interactions with enterprise databases and transaction systems. You can run the DB2 server on the same machine as WebSphere Application Server or on a different Web server.

WebSphere Application Server Advanced Edition (not provided with DB2 Universal Developer's Edition) provides additional support for Enterprise JavaBean applications. DB2 Universal Database™ is provided with the WebSphere Application Server Advanced Edition, to be used as the administrative server repository. It introduces server capabilities for applications built to the EJB Specification from Sun Microsystems, which provides support for integrating Web applications to non-Web business systems.

Related concepts:

- "Enterprise Java Beans" on page 483

Related reference:

- "Java WebSphere samples" in the *Application Development Guide: Building and Running Applications*

XML Extender

Extensible Markup Language (XML) is the accepted standard technique for data exchange between applications. An XML document is a tagged document which is human-legible. The text consists of character data and markup tags. The markup tags are definable by the author of the document. A Document Type Definition (DTD) is used to declare the markup definitions and constraints. DB2® XML Extender (provided with DB2 Universal Developer's Edition, as well as with Personal Developer's Edition on Windows®) gives a mechanism for programs to manipulate XML data using SQL extensions.

The DB2 XML Extender introduces three new data types: `XMLVARCHAR`, `XMLCLOB`, and `XMLFILE`. The extender provides UDFs to store, extract and update XML documents located within single or multiple columns and tables. Searching can be performed on the entire XML document or based on structural components using the location path, which uses a subset of the Extensible Stylesheet Language Transformation (XSLT) and XPath for XML Path Language.

To facilitate storing XML documents as a set of columns, the DB2 XML Extender provides an administration tool to aid the designer with XML-to-relational database mapping. The Document Access Definition (DAD) is used to maintain the structural and mapping data for the XML documents. The DAD is defined and stored as an XML document, which makes it simple to manipulate and understand. New stored procedures are available to compose or decompose the document.

For more information on DB2 XML Extender, visit:

`http://www.ibm.com/software/data/db2/extenders/xmlext/index.html`

MQSeries Enablement

A set of MQSeries® functions are provided with DB2® Universal Database to allow DB2 applications to interact with asynchronous messaging operations. This means that MQSeries support is available to applications written in any programming language supported by DB2.

In a basic configuration, an MQSeries server is located on the database server machine along with DB2 Universal Database™. The MQSeries functions are available from a DB2 server and provide access to other MQSeries applications. Multiple DB2 clients can concurrently access the MQSeries functions through the database. The MQSeries operations allow DB2 applications to asynchronously communicate with other MQSeries applications. For instance, the new functions provide a simple way for a DB2 application to publish database events to remote MQSeries applications, initiate a workflow through the optional MQSeries Workflow product, or communicate with an existing application package with the optional MQSeries Integrator product.

Net.Data

Net.Data® enables Internet and intranet access to DB2® data through your web applications. It exploits Web server interfaces (APIs), providing higher performance than common gateway interface (CGI) applications. Net.Data supports client-side processing as well as server-side processing with languages such as Java™, REXX, Perl and C++. Net.Data provides conditional logic and a rich macro language. It

also provides XML support which allows you to generate XML tags as output from your Net.Data macro, instead of manually entering the tags. You can also specify an XML style sheet (XSL) to be used to format and display the generated output. Net.Data is only available as a Web-based download. For more information, refer to the following Web site:

```
http://www-4.ibm.com/software/data/net.data/support/index.html
```

Note: Net.Data support stabilized in DB2 Version 7.2, and no enhancements for Net.Data support are planned for the future.

Related concepts:

- "Tools for Building Web Applications" on page 14
- "XML Extender" on page 16

Programming Features

The sections that follow describe the programming features that are available with DB2.

DB2 Programming Features

DB2® comes with a variety of features that run on the server which you can use to supplement or extend your applications. When you use DB2 features, you do not have to write your own code to perform the same tasks. DB2 also lets you store some parts of your code at the server instead of keeping all of it in your client application. This can have performance and maintenance benefits.

There are features to protect data and to define relationships between data. As well, there are object-relational features to create flexible, advanced applications. You can use some features in more than one way. For example, constraints enable you to protect data and to define relationships between data values. Here are some key DB2 features:

- Constraints
- User-defined types (UDTs) and large objects (LOBs)
- User-defined functions (UDFs)
- Triggers
- Stored procedures

To decide whether or not to use DB2 features, consider the following points:

Application independence
: You can make your application independent of the data it processes. Using DB2 features that run at the database enables you to maintain and change the logic surrounding the data without affecting your application. If you need to make a change to that logic, you only need to change it in one place; at the server, and not in each application that accesses the data.

Performance
: You can make your application perform more quickly by storing and running parts of your application on the server. This shifts some processing to generally more powerful server machines, and can reduce network traffic between your client application and the server.

Application requirements

Your application might have unique logic that other applications do not. For example, if your application processes data entry errors in a particular order that would be inappropriate for other applications, you might want to write your own code to handle this situation.

In some cases, you might decide to use DB2 features that run on the server because they can be used by several applications. In other cases, you might decide to keep logic in your application because it is used by your application only.

Related concepts:

- "DB2 Stored Procedures" on page 18
- "DB2 User-Defined Functions and Methods" on page 19
- "User-Defined Types (UDTs) and Large Objects (LOBs)" on page 20
- "DB2 Triggers" on page 22

DB2 Stored Procedures

Typically, applications access the database across the network. This can result in poor performance if a lot of data is being returned. A stored procedure runs on the database server. A client application can call the stored procedure which then performs the database accessing without returning unnecessary data across the network. Only the results the client application needs are returned by the stored procedure.

You gain several benefits using stored procedures:

Reduced network traffic

Grouping SQL statements together can save on network traffic. A typical application requires two trips across the network for each SQL statement. Grouping SQL statements results in two trips across the network for each group of statements, resulting in better performance for applications.

Access to features that exist only on the server

Stored procedures can have access to commands that run only on the server, such as LIST DATABASE DIRECTORY and LIST NODE DIRECTORY; they might have the advantages of increased memory and disk space on server machines; and they can access any additional software installed on the server.

Enforcement of business rules

You can use stored procedures to define business rules that are common to several applications. This is another way to define business rules, in addition to using constraints and triggers.

When an application calls the stored procedure, it will process data in a consistent way according to the rules defined in the stored procedure. If you need to change the rules, you only need to make the change once in the stored procedure, not in every application that calls the stored procedure.

Related concepts:

- "Development Center" on page 19

DB2 User-Defined Functions and Methods

The built-in capabilities supplied through SQL may not satisfy all of your application needs. To allow you to extend those capabilities, DB2® supports user-defined functions (UDFs) and methods. You can write your own code in Visual Basic, C/C++, Java™, or SQL to perform operations within any SQL statement that returns a single scalar value or a table.

UDFs and methods give you significant flexibility. They return a single scalar value as part of an expression. Additionally, functions can return whole tables from non-database sources such as spreadsheets.

UDFs and methods provide a way to standardize your applications. By implementing a common set of routines, many applications can process data in the same way, thus ensuring consistent results.

User-defined functions and methods also support object-oriented programming in your applications. They provide for abstraction, allowing you to define the common interfaces that can be used to perform operations on data objects. And they provide for encapsulation, allowing you to control access to the underlying data of an object, protecting it from direct manipulation and possible corruption.

Development Center

DB2® Development Center provides an easy-to-use development environment for creating, installing, and testing stored procedures. It allows you to focus on creating your stored procedure logic rather than the details of registering, building, and installing stored procedures on a DB2 server. Additionally, with Development Center, you can develop stored procedures on one operating system and build them on other server operating systems.

Development Center is a graphical application that supports rapid development. Using Development Center, you can perform the following tasks:

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

You can launch Development Center as a separate application from the DB2 Universal Database™ program group, or you can launch Development Center from any of the following development applications:

- Microsoft® Visual Studio
- Microsoft Visual Basic
- IBM® VisualAge® for Java™

You can also launch Development Center from the Control Center for DB2 for OS/390®. You can start Development Center as a separate process from the Control Center Tools menu, toolbar, or Stored Procedures folder. In addition, from the Development Center 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).

Development Center manages your work by using projects. Each Development Center 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 Development Center project, you can filter stored procedures so that you view stored procedures based on their name, schema, language, or collection ID (for OS/390 only).

Connection information is saved in a Development Center project; therefore, when you open an existing project, you are automatically prompted to enter your user ID and password for the database. Using the Inserting SQL Stored Procedure wizard, you can build SQL stored procedures on a DB2 for OS/390 server. For an SQL stored procedure built to a DB2 for OS/390 server, you can set specific compile, pre-link, link, bind, runtime, WLM environment, and external security options.

Additionally, you can obtain SQL costing information about the SQL stored procedure, including information about CPU time and other DB2 costing information for the thread on which the SQL stored procedure is running. In particular, you can obtain costing information about latch/lock contention wait time, the number of getpages, the number of read I/Os, and the number of write I/Os.

To obtain costing information, Development Center connects to a DB2 for OS/390 server, executes the SQL statement, and calls a stored procedure (DSNWSPM) to find out how much CPU time the SQL stored procedure used.

Related concepts:

- "DB2 Stored Procedures" on page 18
- "OLE Automation Routines" on page 21

User-Defined Types (UDTs) and Large Objects (LOBs)

Every data element in the database is stored in a column of a table, and each column is defined to have a data type. The data type places limits on the types of values you can put into the column and the operations you can perform on them. For example, a column of integers can only contain numbers within a fixed range. DB2® includes a set of built-in data types with defined characteristics and behaviors: character strings, numerics, datetime values, large objects, Nulls, graphic strings, binary strings, and datalinks.

Sometimes, however, the built-in data types might not serve the needs of your applications. DB2 provides user-defined types (UDTs) which enable you to define the distinct data types you need for your applications.

UDTs are based on the built-in data types. When you define a UDT, you also define the operations that are valid for the UDT. For example, you might define a MONEY data type that is based on the DECIMAL data type. However, for the MONEY data type, you might allow only addition and subtraction operations, but not multiplication and division operations.

Large Objects (LOBs) enable you to store and manipulate large, complex data objects in the database: objects such as audio, video, images, and large documents.

The combination of UDTs and LOBs gives you considerable power. You are no longer restricted to using the built-in data types provided by DB2 to model your business data, and to capture the semantics of that data. You can use UDTs to define large, complex data structures for advanced applications.

In addition to extending built-in data types, UDTs provide several other benefits:

Support for object-oriented programming in your applications
You can group similar objects into related data types. These types have a name, an internal representation, and a specific behavior. By using UDTs, you can tell DB2 the name of your new type and how it is represented internally. A LOB is one of the possible internal representations for your new type, and is the most suitable representation for large, complex data structures.

Data integrity through strong typing and encapsulation
Strong typing guarantees that only functions and operations defined on the distinct type can be applied to the type. Encapsulation ensures that the behavior of UDTs is restricted by the functions and operators that can be applied to them. In DB2, behavior for UDTs can be provided in the form of user-defined functions (UDFs), which can be written to accommodate a broad range of user requirements.

Performance through integration into the database manager
Because UDTs are represented internally, the same way as built-in data types, they share the same efficient code as built-in data types to implement built-in functions, comparison operators, indexes, and other functions. The exception to this is UDTs that utilize LOBs, which cannot be used with comparison operators and indexes.

Related concepts:

- "Large object usage" in the *Application Development Guide: Programming Server Applications*
- "User-Defined Types" in the *Application Development Guide: Programming Server Applications*

OLE Automation Routines

OLE (Object Linking and Embedding) automation is part of the OLE 2.0 architecture from Microsoft® Corporation. With OLE automation, your applications, regardless of the language in which they are written, can expose their properties and methods in OLE automation objects. Other applications, such as Lotus® Notes or Microsoft Exchange, can then integrate these objects by taking advantage of these properties and methods through OLE automation.

DB2® for Windows® operating systems provides access to OLE automation objects using UDFs, methods, and stored procedures. To access OLE automation objects and invoke their methods, you must register the methods of the objects as routines (UDFs, methods, or stored procedures) in the database. DB2 applications can then use the methods by invoking the routines.

For example, you can develop an application that queries data in a spreadsheet created using a product such as Microsoft Excel. To do this, you would develop an OLE automation table function that retrieves data from the worksheet, and returns it to DB2. DB2 can then process the data, perform online analytical processing (OLAP), and return the query result to your application.

Related concepts:

- "DB2 Stored Procedures" on page 18
- "Development Center" on page 19

OLE DB Table Functions

Microsoft® OLE DB is a set of OLE/COM interfaces that provide applications with uniform access to data stored in diverse information sources. DB2® Universal Database simplifies the creation of OLE DB applications by enabling you to define table functions that access an OLE DB data source. You can perform operations including GROUP BY, JOIN, and UNION, on data sources that expose their data through OLE DB interfaces. For example, you can define an OLE DB table function to return a table from a Microsoft Access database or a Microsoft Exchange address book, then create a report that seamlessly combines data from this OLE DB table function with data in your DB2 database.

Using OLE DB table functions reduces your application development effort by providing built-in access to any OLE DB provider. For C, Java™, and OLE automation table functions, the developer needs to implement the table function, whereas in the case of OLE DB table functions, a generic built-in OLE DB consumer interfaces with any OLE DB provider to retrieve data. You only need to register a table function of language type OLEDB, and refer to the OLE DB provider and the relevant rowset as a data source. You do not have to do any UDF programming to take advantage of OLE DB table functions.

Related concepts:

- "Purpose of the IBM OLE DB Provider for DB2" on page 219
- "OLE DB Services Automatically Enabled by IBM OLE DB Provider" on page 222

Related reference:

- "IBM OLE DB Provider Support for OLE DB Components and Interfaces" on page 227
- "IBM OLE DB Provider support for OLE DB properties" on page 230

DB2 Triggers

A trigger defines a set of actions executed executed in response to the event of an INSERT, UPDATE or DELETE operation on a specified table. When such an SQL operation is executed, the trigger is said to be activated. The trigger can be activated before the SQL operation or after it. You define a trigger using the SQL statement CREATE TRIGGER.

You can use triggers that run before an update or insert in several ways:

- To check or modify values before they are actually updated or inserted in the database. This is useful if you need to transform data from the way the user sees it to some internal database format.
- To run other non-database operations coded in user-defined functions.

Similarly, you can use triggers that run after an update or insert in several ways:

- To update data in other tables. This capability is useful for maintaining relationships between data or in keeping audit trail information.
- To check against other data in the table or in other tables. This capability is useful to ensure data integrity when referential integrity constraints aren't appropriate, or when table check constraints limit checking to the current table only.

- To run non-database operations coded in user-defined functions. This capability is useful when issuing alerts or to update information outside the database.

You gain several benefits using triggers:

Faster application development
Triggers are stored in the database, and are available to all applications, which relieves you of the need to code equivalent functions for each application.

Global enforcement of business rules
Triggers are defined once, and are used by all applications that use the data governed by the triggers.

Easier maintenance
Any changes need to be made only once in the database instead of in every application that uses a trigger.

Related concepts:

- "Triggers in application development" in the *Application Development Guide: Programming Server Applications*
- "Trigger creation guidelines" in the *Application Development Guide: Programming Server Applications*

Chapter 2. Coding a DB2 Application

Prerequisites for Programming

Before developing an application, you require the appropriate operating environment. The following must also be properly installed and configured:

- A supported compiler or interpreter for developing your applications.
- DB2 Universal Database, either locally or remotely.
- DB2 Application Development Client.

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

DB2® supports the C, C++, Java (SQLJ), COBOL, and FORTRAN programming languages through its precompilers. In addition, DB2 provides support for the Perl, Java (JDBC), and REXX dynamically interpreted languages

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

DB2 provides a sample database, which you require to run the supplied sample programs.

Related tasks:

- "Setting up the application development environment" in the *Application Development Guide: Building and Running Applications*
- "Setting up the sample database" in the *Application Development Guide: Building and Running Applications*

DB2 Application Coding Overview

The sections that follow provide an overview of coding a DB2 application.

Programming a Standalone Application

A standalone application is an application that does not call database objects, such as stored procedures, when it executes. When you write the application, you must ensure that certain SQL statements appear at the beginning and end of the program to handle the transition from the host language to the embedded SQL statements.

Procedure:

To program a standalone application, you must ensure that you:

1. Create the declaration section.
2. Connect to the database.
3. Write one or more transactions.
4. End each transaction using either of the following methods:
   - Commit the changes made by the application to the database.
   - Roll back the changes made by the application to the database.
5. End the program.

Related concepts:

- "Prerequisites for Programming" on page 25
- "Application Pseudocode Framework" on page 38
- "Facilities for Prototyping SQL Statements" on page 39
- "Sample files" in the *Application Development Guide: Building and Running Applications*

Related tasks:

- "Creating the Declaration Section of a Standalone Application" on page 27
- "Connecting an Application to a Database" on page 33
- "Coding Transactions" on page 34
- "Ending a Transaction with the COMMIT Statement" on page 35
- "Ending a Transaction with the ROLLBACK Statement" on page 36
- "Ending an Application Program" on page 37
- "Setting up a Testing Environment" on page 49

Creating the Declaration Section of a Standalone Application

The beginning of every program must contain a declaration section, which contains:

- Declarations of all variables and data structures that the database manager uses to interact with the host program
- SQL statements that provide for error handling by setting up the SQL Communications Area (SQLCA)

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

A program may contain multiple SQL declare sections.

Procedure:

To create the declaration section:

1. Use the SQL statement BEGIN DECLARE SECTION to open the section.
2. Code your declarations
3. Use the SQL statement END DECLARE SECTION to end the section.

Related tasks:

- "Declaring Variables That Interact with the Database Manager" on page 27
- "Declaring Variables That Represent SQL Objects" on page 28
- "Relating Host Variables to an SQL Statement" on page 30
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Declaring the SQLCA for Error Handling" on page 31

Declaring Variables That Interact with the Database Manager

All variables that interact with the database manager must be declared in the SQL declare section.

Host program variables declared in an SQL declare section are called host variables. You can use host variables in host-variable references in SQL statements. The *host-variable* tag is used in syntax diagrams in SQL statements.

Procedure:

To declare a variable, code it in the SQL declare section. An example of a host variable in C/C++ is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
  short     dept=38, age=26;
  double    salary;
  char      CH;
  char      name1[9], NAME2[9];
  /* C comment */
  short     nul_ind;
EXEC SQL END DECLARE SECTION;
```

The attributes of each host variable depend on how the variable is used in the SQL statement. For example, variables that receive data from or store data in DB2 tables must have data type and length attributes compatible with the column being accessed. To determine the data type for each variable, you must be familiar with DB2 data types.

Related reference:

- "Supported SQL Data Types in C and C++" on page 162
- "Supported SQL Data Types in COBOL" on page 190
- "Supported SQL Data Types in FORTRAN" on page 206
- "Supported SQL Data Types in REXX" on page 502
- "Java, JDBC, and SQL data types" on page 365

Declaring Variables That Represent SQL Objects

Declare the variables that represent SQL objects in the SQL declare section of your application program.

Procedure:

Code the variable in the appropriate format for the language in which you are writing your application program.

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

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

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

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

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

For C/C++ applications, you can simplify the coding of declarations and increase the clarity of your code by using C macro expansion to declare the lengths of SQL object identifiers. Because the include file `sql.h` declares SQL_MAX_IDENT to be 128, you can easily declare SQL object identifiers with the SQL_MAX_IDENT macro. For example:

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

Related concepts:

- "Host Variables in C and C++" on page 137
- "Syntax for Fixed and Null-Terminated Character Host Variables in C and C++" on page 140
- "C Macro Expansion" on page 149
- "Host Variables in COBOL" on page 180
- "Host Variables in FORTRAN" on page 200
- "Host Variables in REXX" on page 497

Related reference:

- "Syntax for Numeric Host Variables in C and C++" on page 139
- "Syntax for Variable-Length Character Host Variables in C or C++" on page 141
- "Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++" on page 143
- "Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++" on page 145
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 146
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 147
- "Syntax for File Reference Host Variable Declarations in C or C++" on page 148
- "Syntax for Numeric Host Variables in COBOL" on page 181
- "Syntax for Fixed-Length Character Host Variables in COBOL" on page 182
- "Syntax for Fixed-Length Graphic Host Variables in COBOL" on page 183
- "Syntax for LOB Host Variables in COBOL" on page 184
- "Syntax for LOB Locator Host Variables in COBOL" on page 185
- "Syntax for File Reference Host Variables in COBOL" on page 186
- "Syntax for Numeric Host Variables in FORTRAN" on page 202
- "Syntax for Character Host Variables in FORTRAN" on page 202
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 204
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 205
- "Syntax for File Reference Host Variables in FORTRAN" on page 205
- "Syntax for LOB Locator Declarations in REXX" on page 500
- "Syntax for LOB File Reference Declarations in REXX" on page 501

Declaring Host Variables with the db2dclgn Declaration Generator

You can use the Declaration Generator to generate declarations for a given table in a database. It creates embedded SQL declaration source files which you can easily insert into your applications. `db2dclgn` supports the C/C++, Java, COBOL, and FORTRAN languages.

Procedure:

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

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

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

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

The resulting `staff.h` file contains:

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[6];
  short years;
  double salary;
  double comm;
} staff;
```

Related reference:

- "db2dclgn - Declaration Generator Command" in the *Command Reference*

Relating Host Variables to an SQL Statement

You use host variables to receive data from the database manager or to transfer data to it from the host program. Host variables that receive data from the database manager are *output host variables*, while those that transfer data to it from the host program are *input host variables*.

Consider the following SELECT INTO statement:

```
SELECT HIREDATE, EDLEVEL
       INTO :hdate, :lvl
       FROM EMPLOYEE
       WHERE EMPNO = :idno
```

The statement contains two output host variables, `hdate` and `lvl`, and one input host variable, `idno`. The database manager uses the data stored in the host variable `idno` to determine the EMPNO of the row that is retrieved from the EMPLOYEE table. If the database manager finds a row that meets the search criteria, `hdate` and `lvl` receive the data stored in the columns HIREDATE and EDLEVEL, respectively. This statement illustrates an interaction between the host program and the database manager using columns of the EMPLOYEE table.

Procedure:

To define the host variable for use with a column:

1. Find out the SQL data type for that column. Do this by querying the system catalog, which is a set of views containing information about all tables created in the database.
2. Code the appropriate declarations based on the host language.

   Each column of a table is assigned a data type in the CREATE TABLE definition. You must relate this data type to the host language data type. For example, the INTEGER data type is a 32-bit signed integer. This is equivalent to the following data description entries in each of the host languages, respectively:

C/C++:

```
sqlint32 variable_name;
```

Java:

```
int variable_name;
```

COBOL:

```
01  variable-name  PICTURE S9(9) COMPUTATIONAL-5.
```

FORTRAN:

```
INTEGER*4 variable_name
```

You can also use the Declaration Generator utility (`db2dclgn`) to generate the appropriate declarations for a given table in a database.

Related concepts:

- "Catalog views" in the *SQL Reference, Volume 1*

Related tasks:

- "Declaring Variables That Interact with the Database Manager" on page 27
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Creating the Declaration Section of a Standalone Application" on page 27

Related reference:

- "Supported SQL Data Types in C and C++" on page 162
- "Supported SQL Data Types in COBOL" on page 190
- "Supported SQL Data Types in FORTRAN" on page 206
- "Supported SQL Data Types in REXX" on page 502
- "Java, JDBC, and SQL data types" on page 365

Declaring the SQLCA for Error Handling

You can declare the SQLCA in your application program so that the database manager can return information to your application. When you preprocess your program, the database manager inserts host language variable declarations in place of the INCLUDE SQLCA statement. The system communicates with your program using the variables for warning flags, error codes, and diagnostic information.

After executing each SQL statement, the system returns a return code in both SQLCODE and SQLSTATE. SQLCODE is an integer value that summarizes the execution of the statement, and SQLSTATE is a character field that provides common error codes across IBM's relational database products. SQLSTATE also conforms to the ISO/ANS SQL92 and FIPS 127-2 standard.

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

Note that if SQLCODE is less than 0, it means an error has occurred and the statement has not been processed. If the SQLCODE is greater than 0, it means a warning has been issued, but the statement is still processed.

For a DB2 application written in C or C++, if the application is made up of multiple source files, only one of the files should include the EXEC SQL INCLUDE

SQLCA statement to avoid multiple definitions of the SQLCA. The remaining source files should use the following lines:

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

Procedure:

To declare the SQLCA, code the `INCLUDE SQLCA` statement in your program as follows:

- For C or C++ applications use:

  ```
  EXEC SQL INCLUDE SQLCA;
  ```

- For Java applications, you do not explicitly use the SQLCA. Instead, use the `SQLException` instance methods to get the SQLSTATE and SQLCODE values.
- For COBOL applications use:

  ```
  EXEC SQL INCLUDE SQLCA END-EXEC.
  ```

- For FORTRAN applications use:

  ```
  EXEC SQL INCLUDE SQLCA
  ```

If your application must be compliant with the ISO/ANS SQL92 or FIPS 127-2 standard, do not use the above statements or the `INCLUDE SQLCA` statement.

Related concepts:

- "Error Handling Using the WHENEVER Statement" on page 32
- "SQLSTATE and SQLCODE Variables in C and C++" on page 168
- "SQLSTATE and SQLCODE Variables in COBOL" on page 193
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 208
- "SQLSTATE and SQLCODE Variables in Perl" on page 491

Related tasks:

- "Creating the Declaration Section of a Standalone Application" on page 27

Error Handling Using the WHENEVER Statement

The WHENEVER statement causes the precompiler to generate source code that directs the application to go to a specified label if either an error, a warning, or no rows are found during execution. The WHENEVER statement affects all subsequent executable SQL statements until another WHENEVER statement alters the situation.

The WHENEVER statement has three basic forms:

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

In the above statements:

SQLERROR
: Identifies any condition where SQLCODE < `0`.

SQLWARNING
: Identifies any condition where SQLWARN(0) = `W` or SQLCODE > `0` but is not equal to `100`.

NOT FOUND
Identifies any condition where SQLCODE = 100.

In each case, the *action* can be either of the following:

CONTINUE
Indicates to continue with the next instruction in the application.

GO TO *label*
Indicates to go to the statement immediately following the label specified after GO TO. (GO TO can be two words, or one word, GOTO.)

If the WHENEVER statement is not used, the default action is to continue processing if an error, warning, or exception condition occurs during execution.

The WHENEVER statement must appear before the SQL statements you want to affect. Otherwise, the precompiler does not know that additional error-handling code should be generated for the executable SQL statements. You can have any combination of the three basic forms active at any time. The order in which you declare the three forms is not significant.

To avoid an infinite looping situation, ensure that you undo the WHENEVER handling before any SQL statements are executed inside the handler. You can do this using the WHENEVER SQLERROR CONTINUE statement.

Related reference:

- "WHENEVER statement" in the *SQL Reference, Volume 2*

Adding Non-Executable Statements to an Application

If you need to include non-executable SQL statements in an application program, you typically put them in the declaration section of the application. Examples of non-executable statements are the INCLUDE, INCLUDE SQLDA, and DECLARE CURSOR statements.

Procedure:

If you want to use the non-executable statement INCLUDE in your application, code it as follows:

```
INCLUDE text-file-name
```

Related tasks:

- "Creating the Declaration Section of a Standalone Application" on page 27

Connecting an Application to a Database

Your program must establish a connection to the target database before it can run any executable SQL statements. This connection identifies both the authorization ID of the user who is running the program, and the name of the database server on which the program is run. Generally, your application process can only connect to one database server at a time. This server is called the *current server*. However, your application can connect to multiple database servers within a multisite update environment. In this case, only one server can be the current server.

Restrictions:

The following restrictions apply:

- A connection lasts until a CONNECT RESET, CONNECT TO, or DISCONNECT statement is issued.
- In a multisite update environment, a connection also lasts until a DB2 RELEASE then DB2 COMMIT is issued. A CONNECT TO statement does not terminate a connection when using multisite update.

Procedure:

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

- Explicitly, using the CONNECT statement
- Implicitly, connecting to the default database server
- For Java applications, through a `Connection` instance

See the CONNECT statement description for a discussion of connection states and how to use the CONNECT statement. Upon initialization, the application requester establishes a default database server. If implicit connects are enabled, application processes started after initialization connect implicitly to the default database server. It is good practice to use the CONNECT statement as the first SQL statement executed by an application program. An explicit CONNECT avoids accidentally executing SQL statements against the default database.

Related concepts:

- "Multisite Update" on page 627

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*

Coding Transactions

A transaction is a sequence of SQL statements (possibly with intervening host language code) that the database manager treats as a whole. An alternative term that is often used for transaction is *unit of work*.

Prerequisites:

A connection must be established with the database against which the transaction will execute.

Procedure:

To code a transaction:

1. Start the transaction with an *executable* SQL statement.

   After the connection to the database is established, your program can issue one or more:

   - Data manipulation statements (for example, the SELECT statement)
   - Data definition statements (for example, the CREATE statement)
   - Data control statements (for example, the GRANT statement)

   An executable SQL statement always occurs within a transaction. If a program contains an executable SQL statement after a transaction ends, it automatically starts a new transaction.

Note: The following six statements do *not* start a transaction because they are not executable statements:

- BEGIN DECLARE SECTION
- INCLUDE SQLCA
- END DECLARE SECTION
- INCLUDE SQLDA
- DECLARE CURSOR
- WHENEVER

2. End the transaction in either of the following ways:
   - COMMIT the transaction
   - ROLLBACK the transaction

Related tasks:

- "Ending a Transaction with the COMMIT Statement" on page 35
- "Ending a Transaction with the ROLLBACK Statement" on page 36

Ending a Transaction with the COMMIT Statement

The COMMIT statement ends the current transaction and makes the database changes performed during the transaction visible to other processes.

Procedure:

Commit changes as soon as application requirements permit. In particular, write your programs so that uncommitted changes are not held while waiting for input from a terminal, as this can result in database resources being held for a long time. Holding these resources prevents other applications that need these resources from running.

Your application programs should explicitly end any transactions before terminating.

If you do not end transactions explicitly, DB2 automatically commits all the changes made during the program's pending transaction when the program ends successfully, except on Windows operating systems. On Windows operating systems, if you do not explicitly commit the transaction, the database manager always rolls back the changes.

DB2 rolls back the changes under the following conditions:

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

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

Related concepts:

- "Implicit Ending of a Transaction in a Standalone Application" on page 37
- "Return Codes" on page 99
- "Error Information in the SQLCODE, SQLSTATE, and SQLWARN Fields" on page 100

Related tasks:

- "Ending an Application Program" on page 37

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

Ending a Transaction with the ROLLBACK Statement

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

Procedure:

To prevent the changes that were effected by the transaction from being committed to the database, issue the ROLLBACK statement to end the transaction. The ROLLBACK statement returns the database to the state it was in before the transaction ran.

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

If you use a ROLLBACK statement in a routine that was entered because of an error or warning and you use the SQL WHENEVER statement, then you should specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK. This avoids a program loop if the ROLLBACK fails with an error or warning.

In the event of a severe error, you will receive a message indicating that you cannot issue a ROLLBACK statement. Do not issue a ROLLBACK statement if a severe error occurs such as the loss of communications between the client and server applications, or if the database gets corrupted. After a severe error, the only statement you can issue is a CONNECT statement.

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

You can code one or more transactions within a single application program, and it is possible to access more than one database from within a single transaction. A transaction that accesses more than one database is called a multisite update.

Related concepts:
- "Implicit Ending of a Transaction in a Standalone Application" on page 37
- "Remote Unit of Work" on page 627
- "Multisite Update" on page 627

Related reference:
- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*

- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*
- "WHENEVER statement" in the *SQL Reference, Volume 2*

Ending an Application Program

End an application program to clean up resources that the program was using.

Procedure:

To properly end your program:

1. End the current transaction (if one is in progress) by explicitly issuing either a COMMIT statement or a ROLLBACK statement.
2. Release your connection to the database server by using the CONNECT RESET statement.
3. Clean up resources used by the program. For example, free any temporary storage or data structures that are used.

Note: If the current transaction is still active when the program terminates, DB2 implicitly ends the transaction. Because DB2's behavior when it implicitly ends a transaction is platform specific, you should explicitly end all transactions by issuing a COMMIT or a ROLLBACK statement before the program terminates.

Related concepts:

- "Implicit Ending of a Transaction in a Standalone Application" on page 37

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*

Implicit Ending of a Transaction in a Standalone Application

If your program terminates without ending the current transaction, DB2® implicitly ends the current transaction. DB2 implicitly terminates the current transaction by issuing either a COMMIT or a ROLLBACK statement when the application ends. Whether DB2 issues a COMMIT or ROLLBACK depends on factors such as:

- Whether the application terminated normally

  On most supported operating systems, DB2 implicitly commits a transaction if the termination is normal, or implicitly rolls back the transaction if it is abnormal. Note that what your program considers to be an abnormal termination may not be considered abnormal by the database manager. For example, you may code `exit(-16)` when your application encounters an unexpected error and terminate your application abruptly. The database manager considers this to be a normal termination and commits the transaction. The database manager considers items such as an exception or a segmentation violation as abnormal terminations.
- The platform on which the DB2 server runs

  On Windows® 32-bit operating systems, DB2 always rolls back the transaction regardless of whether your application terminates normally or abnormally. If you want the transaction to be committed, you must issue the COMMIT statement explicitly.

- Whether the application uses the DB2 context APIs for multiple-thread database access

  If your application uses these, DB2 implicitly rolls back the transaction whether your application terminates normally or abnormally. Unless you explicitly commit the transaction using the COMMIT statement, DB2 rolls back the transaction.

Related concepts:

- "Purpose of Multiple-Thread Database Access" on page 169

Related tasks:

- "Ending an Application Program" on page 37

Related reference:

- "COMMIT statement" in the *SQL Reference, Volume 2*
- "ROLLBACK statement" in the *SQL Reference, Volume 2*

Application Pseudocode Framework

The following example summarizes the general framework for a DB2 application program in pseudocode format. You must, of course, tailor this framework to suit your own program.

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

  (program logic)

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

  (more program logic)

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

  (more program logic)

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

  (more program logic)

EXEC SQL CONNECT RESET                                          |
ERRCHK                                                          |
```

```
  (check error information in SQLCA)                    | Application
                                                        | Cleanup
End Program
```

Related tasks:

- "Programming a Standalone Application" on page 26

Facilities for Prototyping SQL Statements

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

- Use the Control Center or the command line processor (CLP) to test many SQL statements before you attempt to compile and link a complete program.

  This allows you to define and manipulate information stored in a database table, index, or view. You can add, delete, or update information as well as generate reports from the contents of tables. Note that you have to minimally change the syntax for some SQL statements in order to use host variables in your embedded SQL program. Host variables are used to store data that is output to your screen. In addition, some embedded SQL statements (such as BEGIN DECLARE SECTION) are not supported by the Command Center or CLP as they are not relevant to that environment.

  You can also redirect the input and output of command line processor requests. For example, you could create one or more files containing SQL statements you need as input into a command line processor request, to save retyping the statement.
- Use the Explain facility to get an idea of the estimated costs of the DELETE, INSERT, UPDATE, or SELECT statements you plan to use in your program. The Explain facility places the information about the structure and the estimated costs of the subject statement into user supplied tables. You can view this information using Visual Explain or the `db2exfmt` utility.
- Use the system catalog views to easily retrieve information about existing databases. The database manager creates and maintains the system catalog tables on which the views are based during normal operation as databases are created, altered, and updated. These views contain data about each database, including authorities granted, column names, data types, indexes, package dependencies, referential constraints, table names, views, and so on. Data in the system catalog views is available through normal SQL query facilities.

  You can update some system catalog views containing statistical information used by the SQL optimizer. You may change some columns in these views to influence the optimizer or to investigate the performance of hypothetical databases. You can use this method to simulate a production system on your development or test system and analyze how queries perform.

Related concepts:

- "Catalog views" in the *SQL Reference, Volume 1*
- "Catalog statistics tables" in the *Administration Guide: Performance*
- "Catalog statistics for modeling and what-if planning" in the *Administration Guide: Performance*
- "General rules for updating catalog statistics manually" in the *Administration Guide: Performance*
- "SQL explain facility" in the *Administration Guide: Performance*

- "DB2 Universal Database tools for developing applications" on page 3

Related reference:

- Appendix A, "Supported SQL Statements," on page 685

Administrative APIs in Embedded SQL or DB2 CLI Programs

Your application can use APIs to access database manager facilities that are not available using SQL statements.

You can use the DB2® APIs to:

- Manipulate the database manager environment, which includes cataloging and uncataloging databases and nodes, and scanning database and node directories. You can also use APIs to create, delete, and migrate databases.
- Provide facilities to import and export data, and administer, backup, and restore the database.
- Modify the database manager and database configuration parameter values.
- Provide operations specific to the client/server environment.
- Provide the run-time interface for precompiled SQL statements. These APIs are not usually called directly by the programmer. Instead, they are inserted into the modified source file by the precompiler after processing.

The database manager includes APIs for language vendors who want to write their own precompiler, and other APIs useful for developing applications.

Related concepts:

- "Authorization Considerations for APIs" on page 48

Controlling Data Values and Relationships

The sections that follow describe how to control data values and data relationships.

Data Value Control

One traditional area of application logic is validating and protecting data integrity by controlling the values allowed in the database. Applications have logic that specifically checks data values as they are entered for validity. (For example, checking that the department number is a valid number and that it refers to an existing department.) There are several different ways of providing these same capabilities in DB2®, but from within the database.

Related concepts:

- "Data Value Control Using Data Types" on page 41
- "Data Value Control Using Unique Constraints" on page 41
- "Data Value Control Using Table Check Constraints" on page 41
- "Data Value Control Using Referential Integrity Constraints" on page 41
- "Data Value Control Using Views with Check Option" on page 42
- "Data Value Control Using Application Logic and Program Variable Types" on page 42

Data Value Control Using Data Types

The database stores every data element in a column of a table, and defines each column with a data type. This data type places certain limits on the types of values for the column. For example, an integer must be a number within a fixed range. The use of the column in SQL statements must conform to certain behaviors; for instance, the database does not compare an integer to a character string. DB2® includes a set of built-in data types with defined characteristics and behaviors. DB2 also supports defining your own data types, called *user-defined distinct types*, that are based on the built-in types but do not automatically support all the behaviors of the built-in type. You can also use data types, like binary large object (BLOB), to store data that may consist of a set of related values, such as a data structure.

Related concepts:

- "User-defined distinct types" in the *Application Development Guide: Programming Server Applications*

Data Value Control Using Unique Constraints

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

Use unique constraints if you need to enforce a uniqueness rule for all applications that use the data in a table.

Related tasks:

- "Defining a unique constraint" in the *Administration Guide: Implementation*
- "Adding a unique constraint" in the *Administration Guide: Implementation*

Data Value Control Using Table Check Constraints

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

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

Related tasks:

- "Defining a table check constraint" in the *Administration Guide: Implementation*
- "Adding a table check constraint" in the *Administration Guide: Implementation*

Data Value Control Using Referential Integrity Constraints

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

updates or deletes that would otherwise result in missing DEPARTMENT information. By centralizing your rules in the database, RI constraints make the rules generally applicable and easier to maintain.

Related concepts:

- "Constraints" in the *SQL Reference, Volume 1*
- "Data Relationship Control Using Referential Integrity Constraints" on page 43
- "Referential Integrity Differences among IBM Relational Database Systems" on page 697

Data Value Control Using Views with Check Option

If your application cannot define the desired rules as table check constraints, or the rules do not apply to all uses of the data, there is another alternative to placing the rules in the application logic. You can consider creating a view of the table with the conditions on the data as part of the WHERE clause and the WITH CHECK OPTION clause specified. This view definition restricts the retrieval of data to the set that is valid for your application. Additionally, if you can update the view, the WITH CHECK OPTION clause restricts updates, inserts, and deletes to the rows applicable to your application.

Related reference:

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

Data Value Control Using Application Logic and Program Variable Types

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

- The rules are not generally applicable, except in the case of views that use the WITH CHECK OPTION
- You do not have control over the definitions of the data in the database
- You believe the rule can be more effectively handled in the application logic

For example, processing errors on input data in the order that they are entered may be required, but cannot be guaranteed from the order of operations within the database.

Related concepts:

- "Data Value Control Using Views with Check Option" on page 42

Data Relationship Control

A major area of focus in application logic is in the area of managing the relationships between different logical entities in your system. For example, if you add a new department, then you need to create a new account code. DB2® provides two methods of managing the relationships between different objects in your database: referential integrity constraints and triggers.

Related concepts:

Data Relationship Control Using Referential Integrity Constraints

Referential integrity (RI) constraints, considered from the perspective of data relationship control, allow you to control the relationships between data in more than one table. Use the CREATE TABLE or ALTER TABLE statements to define the behavior of operations that affect the related primary key, such as DELETE and UPDATE.

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

Related concepts:

- "Constraints" in the *SQL Reference, Volume 1*

Related tasks:

- "Defining referential constraints" in the *Administration Guide: Implementation*

Related reference:

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

Data Relationship Control Using Triggers

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

Related concepts:

- "Data Relationship Control Using Before Triggers" on page 44
- "Data Relationship Control Using After Triggers" on page 44
- "DB2 Triggers" on page 22

Related tasks:

- "Creating a trigger" in the *Administration Guide: Implementation*
- "Creating triggers" in the *Application Development Guide: Programming Server Applications*

Related reference:

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

Data Relationship Control Using Before Triggers

By using triggers that run before an update or insert, values that are being updated or inserted can be modified before the database is actually modified. These can be used to transform input from the application (user view of the data) to an internal database format where desired. These *before triggers* can also be used to cause other non-database operations to be activated through user-defined functions.

Related concepts:

- "Data Relationship Control Using After Triggers" on page 44
- "DB2 Triggers" on page 22

Related tasks:

- "Creating a trigger" in the *Administration Guide: Implementation*
- "Creating triggers" in the *Application Development Guide: Programming Server Applications*

Related reference:

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

Data Relationship Control Using After Triggers

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

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

Related concepts:

- "Data Relationship Control Using Before Triggers" on page 44
- "DB2 Triggers" on page 22

Related tasks:

- "Creating a trigger" in the *Administration Guide: Implementation*
- "Creating triggers" in the *Application Development Guide: Programming Server Applications*

Related reference:

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

Data Relationship Control Using Application Logic

You may decide to write code to enforce rules or perform related operations in the application instead of the database. You must do this for cases where you cannot generally apply the rules to the database. You may also choose to place the logic in

the application when you do not have control over the definitions of the data in the database or you believe the application logic can handle the rules or operations more efficiently.

Related concepts:

- "Application Logic at the Server" on page 45

Application Logic at the Server

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

You can use the following:

- Stored procedures

  A stored procedure is a routine for your application that is called from the client application logic, but runs on the database server. The most common reason to use a stored procedure is for database-intensive processing that produces only small amounts of result data. This can save a large amount of communications across the network during the execution of the stored procedure. You may also consider using a stored procedure for a set of operations that are common to multiple applications. In this way, all the applications use the same logic to perform the operation.
- User-defined functions

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

  – A single scalar value (scalar function)

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

  UDFs are useful for tasks like transforming data values, performing calculations on one or more data values, or extracting parts of a value (such as extracting parts of a large object).
- Triggers

  Triggers can be used to invoke user-defined functions. This is useful when you always want a certain non-SQL operation performed when specific statements occur, or data values are changed. Examples include such operations as issuing an electronic mail message under specific circumstances or writing alert type information to a file.

Related concepts:

- "Data Relationship Control Using Before Triggers" on page 44
- "Data Relationship Control Using After Triggers" on page 44
- "Guidelines for stored procedures" in the *Administration Guide: Performance*
- "Trigger interactions with referential constraints" in the *Application Development Guide: Programming Server Applications*
- "DB2 Stored Procedures" on page 18
- "DB2 User-Defined Functions and Methods" on page 19
- "DB2 Triggers" on page 22

Related tasks:

- "Creating a trigger" in the *Administration Guide: Implementation*
- "Creating triggers" in the *Application Development Guide: Programming Server Applications*

Related reference:

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

Authorization Considerations for SQL and APIs

The sections that follow describe the general authorization considerations for embedded SQL, and the authorization considerations for static and dynamic SQL, and for APIs.

Authorization Considerations for Embedded SQL

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way. DB2® uses a set of privileges to provide protection for the information that you store in it.

Most SQL statements require some type of privilege on the database objects which the statement utilizes. Most API calls usually do not require any privilege on the database objects which the call utilizes, however, many APIs require that you possess the necessary authority in order to invoke them. The DB2 APIs enable you to perform the DB2 administrative functions from within your application program. For example, to recreate a package stored in the database without the need for a bind file, you can use the `sqlarbnd` (or REBIND) API.

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

- Whether your application uses dynamic SQL, including JDBC and DB2 CLI, or static SQL. For information about the privileges required to issue a statement, see the description of that statement.
- Which APIs the application uses. For information about the privileges and authorities required for an API call, see the description of that API.

Consider two users, PAYROLL and BUDGET, who need to perform queries against the STAFF table. PAYROLL is responsible for paying the employees of the company, so it needs to issue a variety of SELECT statements when issuing paychecks. PAYROLL needs to be able to access each employee's salary. BUDGET is responsible for determining how much money is needed to pay the salaries. BUDGET should not, however, be able to see any particular employee's salary.

Because PAYROLL issues many different SELECT statements, the application you design for PAYROLL could probably make good use of dynamic SQL. The dynamic SQL would require that PAYROLL have SELECT privilege on the STAFF table. This requirement is not a problem because PAYROLL requires full access to the table.

BUDGET, on the other hand, should not have access to each employee's salary. This means that you should not grant SELECT privilege on the STAFF table to BUDGET. Because BUDGET does need access to the total of all the salaries in the STAFF table, you could build a static SQL application to execute a SELECT SUM(SALARY) FROM STAFF, bind the application and grant the EXECUTE

privilege on your application's package to BUDGET. This enables BUDGET to obtain the required information, without exposing the information that BUDGET should not see.

Related concepts:

- "Authorization Considerations for Dynamic SQL" on page 47
- "Authorization Considerations for Static SQL" on page 48
- "Authorization Considerations for APIs" on page 48
- "Authorization" in the *Administration Guide: Planning*

Authorization Considerations for Dynamic SQL

To use dynamic SQL in a package bound with DYNAMICRULES RUN (default), the person who runs a dynamic SQL application must have the privileges necessary to issue each SQL request performed, as well as the EXECUTE privilege on the package. The privileges may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

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

If the program contains no static SQL, the person binding the application (for embedded dynamic SQL applications) only needs the BINDADD authority on the database. Again, this privilege can be granted to the user's authorization ID, to a group of which the user is a member, or to PUBLIC.

When a package exhibits bind or define behavior, the user that runs the application needs only the EXECUTE privilege on the package to run it. At run-time, the binder of a package that exhibits bind behavior must have the privileges necessary to execute all the dynamic statements generated by the package, because all authorization checking for dynamic statements is done using the ID of the binder and not the executors. Similarly, the definer of a routine whose package exhibits define behavior must have all the privileges necessary to execute all the dynamic statements generated by the define behavior package. If you have SYSADM or DBADM authority and create a bind behavior package, consider using the OWNER BIND option to designate a different authorization ID. The OWNER BIND option prevents a package from automatically inheriting SYSADM or DBADM privileges within dynamic SQL statements. For more information on the DYNAMICRULES and OWNER bind options, refer to the BIND command. For more information on package behaviors, see the description of DYNAMICRULES effects on dynamic SQL statements.

Related concepts:

- "Authorization Considerations for Embedded SQL" on page 46
- "Authorization Considerations for Static SQL" on page 48
- "Authorization Considerations for APIs" on page 48
- "Authorizations and binding of routines that contain SQL" in the *Application Development Guide: Programming Server Applications*

Related reference:

- "BIND Command" in the *Command Reference*

Authorization Considerations for Static SQL

To use static SQL, the user running the application only needs the EXECUTE privilege on the package. No privileges are required for each of the statements that make up the package. The EXECUTE privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

Unless you specify the VALIDATE RUN option when binding the application, the authorization ID you use to bind the application must have the privileges necessary to perform all the statements in the application. If VALIDATE RUN was specified at BIND time, all authorization failures for any static SQL within this package will not cause the BIND to fail and those statements will be revalidated at run time. The person binding the application must always have BINDADD authority. The privileges needed to execute the statements must be granted to the user's authorization ID or to PUBLIC. Group privileges are not used when binding static SQL statements. As with dynamic SQL, the BINDADD privilege can be granted to the user authorization ID, to a group of which the user is a member, or to PUBLIC.

These properties of static SQL give you very precise control over access to information in DB2®.

Related concepts:

- "Authorization Considerations for Embedded SQL" on page 46
- "Authorization Considerations for Dynamic SQL" on page 47
- "Authorization Considerations for APIs" on page 48

Related reference:

- "BIND Command" in the *Command Reference*

Authorization Considerations for APIs

Most of the APIs provided by DB2® do not require the use of privileges, however, many do require some kind of authority to invoke. For the APIs that do require a privilege, the privilege must be granted to the user running the application. The privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC. For information on the required privilege and authority to issue each API call, see the description of the API.

Some APIs can be accessed via a stored procedure interface. For information whether a specific API can be accessed via a stored procedure, see the description of that API.

Related concepts:

- "Authorization Considerations for Embedded SQL" on page 46
- "Authorization Considerations for Dynamic SQL" on page 47
- "Authorization Considerations for Static SQL" on page 48

Testing the Application

The sections that follow describe how to set up a test environment, and how to debug and optimize the application.

Setting up the Test Environment for an Application

The sections that follow describe how to set up the test environment for your application.

Setting up a Testing Environment

To validate your application, you should set up a test environment. For example, you need a database to test your application's SQL code.

Procedure:

To set up the test environment, do the following:

1. Create a test database.

   To create a test database, write a small server application that calls the CREATE DATABASE API, or use the command line processor.
2. Create test tables and views.

   If your application updates, inserts, or deletes data from tables and views, use test data to verify its execution. If the application only retrieves data from tables and views, consider using production-level data when testing it.
3. Generate test data for the tables.

   The input data used to test an application should be valid data that represents all possible input conditions. If the application verifies that input data is valid, include both valid and invalid data to verify that the valid data is processed and the invalid data is flagged.
4. Debug and optimize the application.

Related tasks:

- "Creating Test Tables and Views" on page 49
- "Generating Test Data" on page 50
- "Debugging and Optimizing an Application" on page 52

Related reference:

- "sqlecrea - Create Database" in the *Administrative API Reference*
- "CREATE DATABASE Command" in the *Command Reference*

Creating Test Tables and Views

To design the test tables and views needed, first analyze the data needs of the application. To create a table, you need the CREATETAB authority and the CREATEIN privilege on the schema. See the CREATE TABLE statement for alternative authorities.

Procedure:

To create test tables:

1. List the data the application accesses and describe how each data item is accessed. For example, suppose the application being developed accesses the TEST.TEMPL, TEST.TDEPT, and TEST.TPROJ tables. You could record the type of accesses as shown in the following table

Table 1. Description of the Application Data

Table or View Name	Insert Rows	Delete Rows	Column Name	Data Type	Update Access
TEST.TEMPL	No	No	EMPNO	CHAR(6)	Yes
			LASTNAME	VARCHAR(15)	Yes
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	
			JOBCODE	DECIMAL(3)	
TEST.TDEPT	No	No	DEPTNO	CHAR(3)	
			MGRNO	CHAR(6)	
TEST.TPROJ	Yes	Yes	PROJNO	CHAR(6)	Yes
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	

2. When the description of the application data access is complete, construct the test tables and views that are needed to test the application:
   - Create a test table when the application modifies data in a table or a view. Create the following test tables using the CREATE TABLE SQL statement:
     - TEMPL
     - TPROJ
   - Create a test view when the application does not modify data in the production database.

     In this example, create a test view of the TDEPT table using the CREATE VIEW SQL statement.
3. Generate test data for the tables.

If the database schema is being developed along with the application, the definitions of the test tables might be refined repeatedly during the development process. Usually, the primary application cannot both create the tables and access them because the database manager cannot bind statements that refer to tables and views that do not exist. To make the process of creating and changing tables less time-consuming, consider developing a separate application to create the tables. You can also create test tables interactively using the command line processor (CLP).

After you complete the procedure, you need to create the related topics for this task.

Related tasks:

- "Generating Test Data" on page 50

Related reference:

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

Generating Test Data

After creating the test tables, you can populate them with test data to verify the data handling behavior of the application.

Procedure:

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

- INSERT...VALUES (an SQL statement) puts one or more rows into a table each time the command is issued.
- INSERT...SELECT obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement.
- The IMPORT or LOAD utility inserts large amounts of new or existing data from a defined source.
- The RESTORE utility can be used to duplicate the contents of an existing database into an identical test database by using a BACKUP copy of the original database.
- The DB2MOVE utility to move large numbers of tables between DB2 databases located on workstations.

The following SQL statements demonstrate a technique you can use to populate your tables with randomly generated test data. Suppose the table EMP contains four columns, ENO (employee number), LASTNAME (last name), HIREDATE (date of hire) and SALARY (employee's salary) as in the following CREATE TABLE statement:

```
CREATE TABLE EMP (ENO INTEGER, LASTNAME VARCHAR(30),
                  HIREDATE DATE, SALARY INTEGER);
```

Suppose you want to populate this table with employee numbers from 1 to a number, say 100, with random data for the rest of the columns. You can do this using the following SQL statement:

```
INSERT INTO EMP
-- generate 100 records
WITH DT(ENO) AS (VALUES(1) UNION ALL
SELECT ENO+1 FROM DT WHERE ENO < 100 ) 1

-- Now, use the generated records in DT to create other columns
-- of the employee record.
SELECT ENO, 2
    TRANSLATE(CHAR(INTEGER(RAND()*1000000)), 3
              CASE MOD(ENO,4) WHEN 0 THEN 'aeiou' || 'bcdfg'
                              WHEN 1 THEN 'aeiou' || 'hjklm'
                              WHEN 2 THEN 'aeiou' || 'npqrs'
                                     ELSE 'aeiou' || 'twxyz' END,
                                          '1234567890') AS LASTNAME,
    CURRENT DATE - (RAND()*10957) DAYS AS HIREDATE, 4
    INTEGER(10000+RAND()*200000) AS SALARY  5
  FROM DT;

SELECT * FROM EMP;
```

The following is an explanation of the above statement:

1. The first part of the INSERT statement generates 100 records for the first 100 employees using a recursive subquery to generate the employee numbers. Each record contains the employee number. To change the number of employees, use a number other than 100.
2. The SELECT statement generates the LASTNAME column. It begins by generating a random integer up to 6 digits long using the RAND function. It then converts the integer to its numeric character format using the CHAR function.
3. To convert the numeric characters to alphabet characters, the statement uses the TRANSLATE function to convert the ten numeric characters (0 through 9) to alphabet characters. Since there are more than 10 alphabet characters, the

statement selects from five different translations. This results in names having enough random vowels to be pronounceable and so the vowels are included in each translation.

4. The statement generates a random HIREDATE value. The value of HIREDATE ranges back from the current date to 30 years ago. HIREDATE is calculated by subtracting a random number of days between 0 and 10 957 from the current date. (10 957 is the number of days in 30 years.)
5. Finally, the statement randomly generates the SALARY. The minimum salary is 10 000, to which a random number from 0 to 200 000 is added.

You may also want to consider prototyping any user-defined functions (UDF) you are developing against the test data.

Related concepts:

- "Import Overview" in the *Data Movement Utilities Guide and Reference*
- "Load Overview" in the *Data Movement Utilities Guide and Reference*
- "DB2 User-Defined Functions and Methods" on page 19

Related tasks:

- "Debugging and Optimizing an Application" on page 52

Related reference:

- "INSERT scalar function" in the *SQL Reference, Volume 1*
- "RESTORE DATABASE Command" in the *Command Reference*

Debugging and Optimizing an Application

You can debug and optimize your application while you develop it.

Procedure:

To debug and optimize your application:

- Prototype your SQL statements. You can use the command line processor, the Explain facility, analyze the system catalog views for information about the tables and databases that your program is manipulating, and update certain system catalog statistics to simulate production conditions.
- Use the flagger facility to check the syntax of SQL statements in applications being developed for DB2 Universal Database for z/OS and OS/390, or for conformance to the SQL92 Entry Level standard. This facility is invoked during precompilation.
- Make full use of the error-handling APIs. For example, you can use error-handling APIs to print all messages during the testing phase.
- Use the database system monitor to capture certain optimizing information for analysis.

Related concepts:

- "Catalog statistics for modeling and what-if planning" in the *Administration Guide: Performance*
- "Facilities for Prototyping SQL Statements" on page 39
- "The database system monitor information" in the *Administration Guide: Performance*
- "Source File Requirements for Embedded SQL Applications" on page 62

Part 2. Embedded SQL

Chapter 3. Embedded SQL Overview

Embedding SQL Statements in a Host Language

You can write applications with SQL statements embedded within a host language. The SQL statements provide the database interface, while the host language provides the remaining support needed for the application to execute.

Procedure:

Use the examples in the following table as a guide on how to embed SQL statements in a host language application. In the example, the application checks the SQLCODE field of the SQLCA structure to determine whether the update was successful.

Table 2. Embedding SQL Statements in a Host Language

Language	Sample Source Code

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

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

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

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

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

- For the C, C++, COBOL, or FORTRAN languages, this conversion is handled by the DB2 precompiler. The DB2 precompiler is invoked using the PREP command. The precompiler converts embedded SQL statements directly into DB2 run-time services API calls.
- For the Java language, the SQLJ translator converts SQLJ clauses into JDBC statements. The SQLJ translator is invoked with the **sqlj** command.

When the precompiler processes a source file, it specifically looks for SQL statements and avoids the non-SQL host language. It can find SQL statements because they are surrounded by special delimiters. The examples in the following table show how to use delimiters and comments to create valid embedded SQL statements in the supported compiled host languages.

Table 3. Embedding SQL Statements in a Host Language

Language	Sample Source Code

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

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

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

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

Related concepts:

- "Embedded SQL in REXX Applications" on page 495
- "Embedded SQL Statements in C and C++" on page 135
- "Embedded SQL Statements in COBOL" on page 178
- "Embedded SQL Statements in FORTRAN" on page 199

Source File Creation and Preparation

You can create the source code in a standard ASCII file, called a source file, using a text editor. The source file must have the proper extension for the host language in which you write your code.

Note: Not all platforms support all host languages.

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

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

Simply stated, precompiling converts embedded SQL statements into DB2 run-time API calls that a host compiler can process, and creates a bind file. The bind file contains information on the SQL statements in the application program. The `BIND` command creates a *package* in the database. Optionally, the precompiler can perform the bind step at precompile time.

Binding is the process of creating a *package* from a bind file and storing it in a database. If your application accesses more than one database, you must create a package for each database.

The following figure shows the order of these steps, along with the various modules of a typical compiled DB2 application.

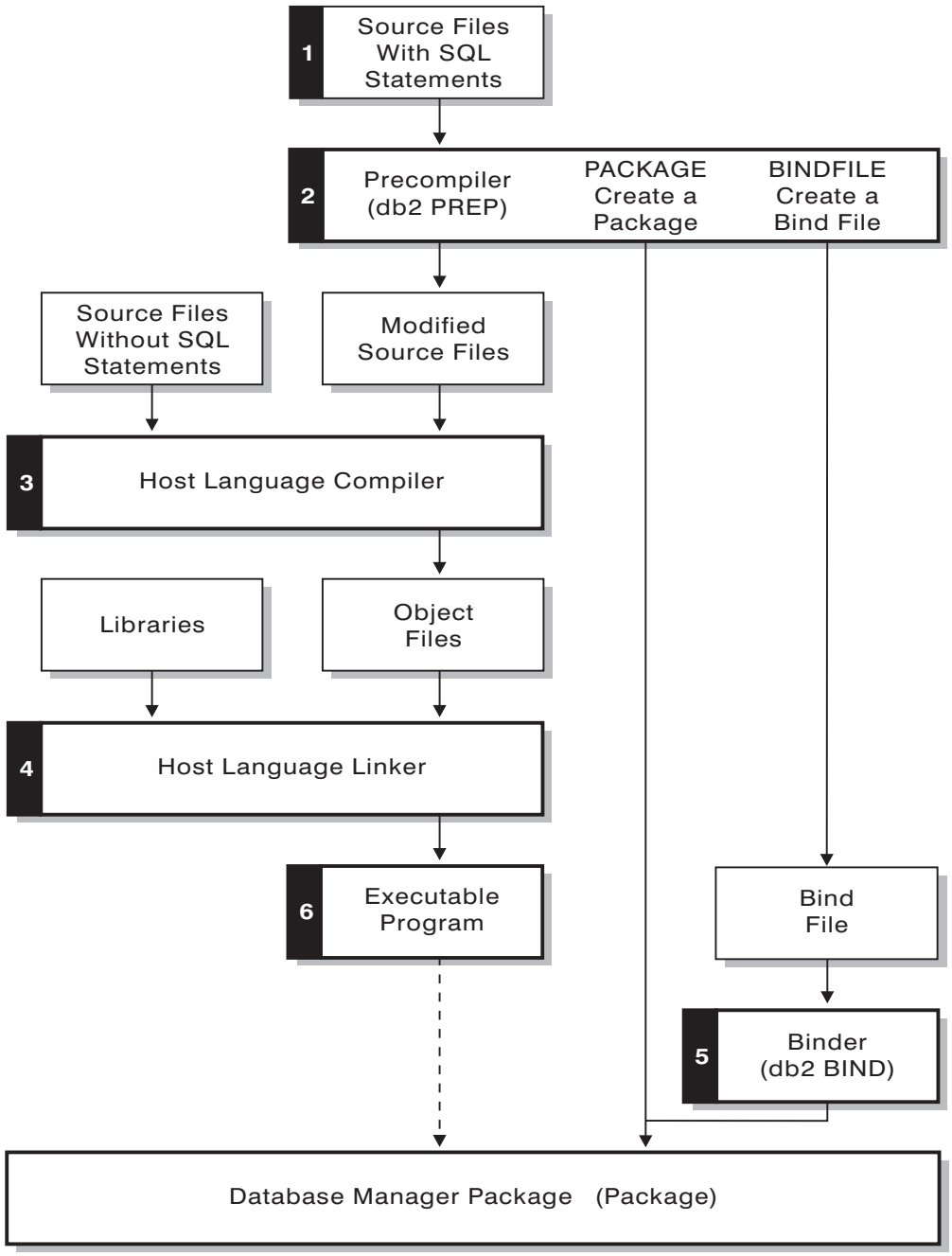


Figure 1. Preparing Programs Written in Compiled Host Languages

Related concepts:

- "Precompilation of Source Files Containing Embedded SQL" on page 61
- "Source File Requirements for Embedded SQL Applications" on page 62
- "Compilation and Linkage of Source Files Containing Embedded SQL" on page 63
- "Embedded SQL" on page 7

Related reference:

- "BIND Command" in the *Command Reference*

Packages, binding, and embedded SQL

The sections that follow describe how to create packages for embedded SQL applications, as well as other topics, such as deferred binding and the relationships between the application, the bind file, and the package.

Package Creation for Embedded SQL

To run applications written in compiled host languages, you must create the packages needed by the database manager at execution time. This involves the following steps as shown in the following figure:

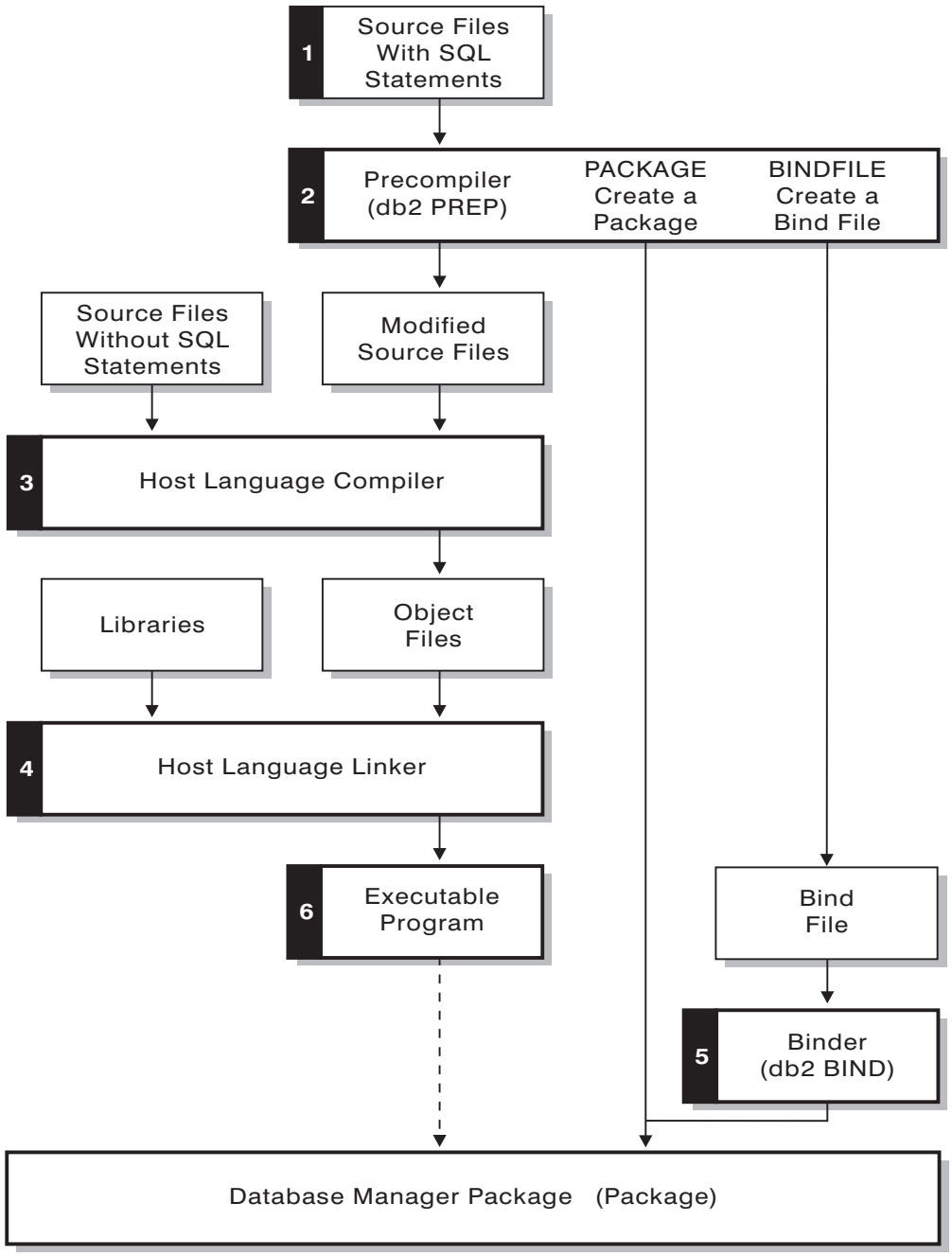


Figure 2. Preparing Programs Written in Compiled Host Languages

- Precompiling (step 2), to convert embedded SQL source statements into a form the database manager can use,
- Compiling and linking (steps 3 and 4), to create the required object modules, and,
- Binding (step 5), to create the package to be used by the database manager when the program is run.

Related concepts:

- "Precompilation of Source Files Containing Embedded SQL" on page 61
- "Source File Requirements for Embedded SQL Applications" on page 62

Related reference:

- "db2bfd - Bind File Description Tool Command" in the *Command Reference*

Precompilation of Source Files Containing Embedded SQL

After you create the source files, you must precompile each host language file containing SQL statements with the `PREP` command for host-language source files. The precompiler converts SQL statements contained in the source file to comments, and generates the DB2 run-time API calls for those statements.

Before precompiling an application you must connect to a server, either implicitly or explicitly. Although you precompile application programs at the client workstation and the precompiler generates modified source and messages on the client, the precompiler uses the server connection to perform some of the validation.

The precompiler also creates the information the database manager needs to process the SQL statements against a database. This information is stored in a package, in a bind file, or in both, depending on the precompiler options selected.

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

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

The precompiler generates up to four types of output:

Modified Source
: This file is the new version of the original source file after the precompiler converts the SQL statements into DB2 run-time API calls. It is given the appropriate host language extension.

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

If you use the PACKAGE option without SQLERROR CONTINUE, the database used during the precompile process must contain all of the database objects referenced by the static SQL statements in the source file. For example, you cannot precompile a SELECT statement unless the table it references exists in the database.

With the VERSION option the bindfile, (if the BINDFILE option is used), and the package (either if bound at PREP time or if a bound separately) will be designated with a particular version identifier. Many versions of packages with the same name and creator can exit at once.

Bind File If you use the BINDFILE option, the precompiler creates a bind file (with extension `.bnd`) that contains the data required to create a package. This file can be used later with the `BIND` command to bind the application to one or more databases. If you specify BINDFILE and do not specify the PACKAGE option, binding is deferred until you invoke the `BIND` command. Note that for the command line processor (CLP), the default for `PREP` does not specify the BINDFILE option. Thus, if you are using the CLP and want the binding to be deferred, you need to specify the BINDFILE option.

Specifying SQLERROR CONTINUE creates a package, even if errors occur when binding SQL statements. Those statements that fail to bind for authorization or existence reasons can be incrementally bound at execution time if VALIDATE RUN is also specified. Any attempt to execute them at run time generates an error.

Message File If you use the MESSAGES option, the precompiler redirects messages to the indicated file. These messages include warnings and error messages that describe problems encountered during precompilation. If the source file does not precompile successfully, use the warning and error messages to determine the problem, correct the source file, and then attempt to precompile the source file again. If you do not use the MESSAGES option, precompilation messages are written to the standard output.

Related concepts:

- "Package Versioning" on page 65

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Source File Requirements for Embedded SQL Applications

You must always precompile a source file against a specific database, even if eventually you do not use the database with the application. In practice, you can use a test database for development, and after you fully test the application, you can bind its bind file to one or more production databases. This practice is known as *deferred binding*.

If your application uses a code page that is not the same as your database code page, you need to consider which code page to use when precompiling.

If your application uses user-defined functions (UDFs) or user-defined distinct types (UDTs), you may need to use the FUNCPATH option when you precompile your application. This option specifies the function path that is used to resolve UDFs and UDTs for applications containing static SQL. If FUNCPATH is not specified, the default function path is *SYSIBM*, *SYSFUN*, *USER*, where *USER* refers to the current user ID.

To precompile an application program that accesses more than one server, you can do one of the following:

- Split the SQL statements for each database into separate source files. Do not mix SQL statements for different databases in the same file. Each source file can be precompiled against the appropriate database. This is the recommended method.
- Code your application using dynamic SQL statements only, and bind against each database your program will access.
- If all the databases look the same, that is, they have the same definition, you can group the SQL statements together into one source file.

The same procedures apply if your application will access a host, AS/400® or iSeries application server through DB2 Connect. Precompile it against the server to which it will be connecting, using the PREP options available for that server.

If you are precompiling an application that will run on DB2 Universal Database for z/OS and OS/390, consider using the flagger facility to check the syntax of the SQL statements. The flagger indicates SQL syntax that is supported by DB2 Universal Database, but not supported by DB2 Universal Database for z/OS and OS/390. You can also use the flagger to check that your SQL syntax conforms to the SQL92 Entry Level syntax. You can use the SQLFLAG option on the PREP command to invoke it and to specify the version of DB2 Universal Database for z/OS and OS/390 SQL syntax to be used for comparison. The flagger facility will not enforce any changes in SQL use; it only issues informational and warning messages regarding syntax incompatibilities, and does not terminate preprocessing abnormally.

Related concepts:

- "Advantages of Deferred Binding" on page 71
- "Character conversion between different code pages" on page 609
- "When code page conversion occurs" on page 609
- "Character Substitutions During Code Page Conversions" on page 610
- "Supported Code Page Conversions" on page 610
- "Code Page Conversion Expansion Factor" on page 611

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Compilation and Linkage of Source Files Containing Embedded SQL

Compile the modified source files and any additional source files that do not contain SQL statements using the appropriate host language compiler. The language compiler converts each modified source file into an *object module*.

Refer to the programming documentation for your operating platform for any exceptions to the default compiler options. Refer to your compiler's documentation for a complete description of available compiler options.

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

- On Windows® operating systems, the application can be an executable file or a dynamic link library (DLL).
- On UNIX®-based systems, the application can be an executable load module or a shared library.

Note: Although applications can be DLLs on Windows operating systems, the DLLs are loaded directly by the application and not by the DB2® database manager. On Windows operating systems, the database manager can load DLLs. Stored procedures are normally built as DLLs or shared libraries.

To create the executable file, link the following:

- User object modules, generated by the language compiler from the modified source files and other files not containing SQL statements.
- Host language library APIs, supplied with the language compiler.
- The database manager library containing the database manager APIs for your operating environment. Refer to the appropriate programming documentation for your operating platform for the specific name of the database manager library you need for your database manager APIs.

Related concepts:

- "DB2 Stored Procedures" on page 18

Related tasks:

- "Building and Running REXX Applications" on page 504
- "Building JDBC applets" in the *Application Development Guide: Building and Running Applications*
- "Building JDBC applications" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ applets" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ applications" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX C applications" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX C++ applications" in the *Application Development Guide: Building and Running Applications*
- "Building IBM COBOL applications on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building UNIX Micro Focus COBOL applications" in the *Application Development Guide: Building and Running Applications*

Package Creation Using the BIND Command

Binding is the process that creates the package the database manager needs to access the database when the application is executed. Binding can be done implicitly by specifying the PACKAGE option during precompilation, or explicitly by using the `BIND` command against the bind file created during precompilation.

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

```
DB2® BIND filename.bnd
```

One package is created for each separately precompiled source code module. If an application has five source files, of which three require precompilation, three packages or bind files are created. By default, each package is given a name that is the same as the name of the source module from which the .bnd file originated, but truncated to 8 characters. To explicitly specify a different package name, you must use the PACKAGE USING option on the PREP command. The version of a package is given by the VERSION precompile option and defaults to the empty string. If the name and schema of this newly created package is the same as a package that currently exists in the target database, but the version identifier differs, a new package is created and the previous package still remains. However if a package exists that matches the name, schema and the version of the package being bound, then that package is dropped and replaced with the new package being bound (specifying ACTION ADD on the bind would prevent that and an error (SQL0719) would be returned instead).

Related reference:

- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

Package Versioning

If you need to create multiple versions of an application, you can use the VERSION option of the PRECOMPILE command. This option allows multiple versions of the same package name (that is, the package name and creator name) to coexist. For example, assume you have an application called foo, which is compiled from foo.sqc. You would precompile and bind the package foo to the database and deliver the application to the users. The users could then run the application. To make subsequent changes to the application, you would update foo.sqc, then repeat the process of recompiling, binding, and sending the application to the users. If the VERSION option was not specified for either the first or second precompilation of foo.sqc, the first package is replaced by the second package. Any user who attempts to run the old version of the application will receive the SQLCODE -818, indicating a mismatched timestamp error.

To avoid the mismatched timestamp error and in order to allow both versions of the application to run at the same time, use package versioning. As an example, when you build the first version of foo, precompile it using the VERSION option, as follows:

```
DB2® PREP FOO.SQC VERSION V1.1
```

This first version of the program may now be run. When you build the new version of foo, precompile it with the command:

```
DB2 PREP FOO.SQC VERSION V1.2
```

At this point this new version of the application will also run, even if there still are instances of the first application still executing. Because the package version for the first package is V1.1 and the package version for the second is V1.2, no naming confict exists: both packages will exist in the database and both versions of the application can be used.

You can use the ACTION option of the PRECOMPILE or BIND commands in conjunction with the VERSION option of the PRECOMPILE command. You use the ACTION option to control the way in which different versions of packages can be added or replaced.

Package privileges do not have granularity at the version level. That is, a GRANT or a REVOKE of a package privilege applies to all versions of a package that share the name and creator. So, if package privileges on package `foo` were granted to a user or a group after version V1.1 was created, when version V1.2 is distributed the user or group has the same privileges on version V1.2. This behavior is usually required because typically the same users and groups have the same privileges on all versions of a package. If you do not want the same package privileges to apply to all versions of an application, you should not use the PRECOMPILE VERSION option to accomplish package versioning. Instead, you should use different package names (either by renaming the updated source file, or by using the PACKAGE USING option to explicitly rename the package).

Related concepts:

- "Precompiler-Generated Timestamps" on page 72

Related reference:

- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

Effect of Special Registers on Bound Dynamic SQL

For dynamically prepared statements, the values of a number of special registers determine the statement compilation environment:

- The CURRENT QUERY OPTIMIZATION special register determines which optimization class is used.
- The CURRENT PATH special register determines the function path used for UDF and UDT resolution.
- The CURRENT EXPLAIN SNAPSHOT register determines whether explain snapshot information is captured.
- The CURRENT EXPLAIN MODE register determines whether explain table information is captured for any eligible dynamic SQL statement. The default values for these special registers are the same defaults used for the related bind options.

Related reference:

- "CURRENT EXPLAIN MODE special register" in the *SQL Reference, Volume 1*
- "CURRENT EXPLAIN SNAPSHOT special register" in the *SQL Reference, Volume 1*
- "CURRENT PATH special register" in the *SQL Reference, Volume 1*
- "CURRENT QUERY OPTIMIZATION special register" in the *SQL Reference, Volume 1*

CURRENT PACKAGE PATH special register for package schemas

Package schemas provide a method for logically grouping packages. Different approaches exist for grouping packages into schemas. Some implementations use one schema per environment (for example, a production and a test schema). Other

implementations use one schema per business area (for example, stocktrd and onlinebnk schemas), or one schema per application (for example, stocktrdAddUser and onlinebnkAddUser). You can also group packages for general administration purposes, or to provide variations in the packages (for example, maintaining backup variations of applications, or testing new variations of applications).

When multiple schemas are used for packages, the database manager must determine in which schema to look for a package. To accomplish this task, the database manager uses the value of the CURRENT PACKAGESET special register. You can set this special register to a single schema name to indicate that any package to be invoked belongs to that schema. If an application uses packages in different schemas, a SET CURRENT PACKAGESET statement might have to be issued before each package is invoked if the schema for the package is different from that of the previous package.

Note: Only DB2® Universal Database for OS/390® and z/OS™ has a CURRENT PACKAGESET special register, which allows you to explicitly set the value (a single schema name) with the corresponding SET CURRENT PACKAGESET statement. Although DB2 Universal Database™ for Linux, UNIX®, and Windows® has a SET CURRENT PACKAGESET statement, it does not have a CURRENT PACKAGESET special register. This means that CURRENT PACKAGESET cannot be referenced in other contexts (such as in a SELECT statement) with DB2 Universal Database for Linux, UNIX, and Windows. DB2 Universal Database for AS/00 does not provide support for CURRENT PACKAGESET.

DB2 has more flexibility when it can consider a list of schemas during package resolution. The list of schemas is similar to the SQL path that is provided by the CURRENT PATH special register. The schema list is used for user-defined functions, procedures, methods, and distinct types.

Note: The SQL path is a list of schema names that DB2 should consider when trying to determine the schema for an unqualified function, procedure, method, or distinct type name.

If you need to associate multiple variations of a package (that is, multiple sets of BIND options for a package) with a single compiled program, consider isolating the path of schemas that are used for SQL objects from the path of schemas that are used for packages.

The CURRENT PACKAGE PATH special register allows you to specify a list of package schemas. Other DB2 family products provide similar capability with special registers such as CURRENT PATH and CURRENT PACKAGESET, which are pushed and popped for nested procedures and user-defined functions without corrupting the runtime environment of the invoking application. The CURRENT PACKAGE PATH special register provides this capability for package schema resolution.

Many installations use more than one schema for packages. If you do not specify a list of package schemas, you must issue the SET CURRENT PACKAGESET statement (which can contain at most one schema name) each time you require a package from a different schema. If, however, you issue a SET CURRENT PACKAGE PATH statement at the beginning of the application to specify a list of schema names, you do not need to issue a SET CURRENT PACKAGESET statement each time a package in a different schema is needed. This capability is especially useful if you are building an SQLJ application, because the application

can search a list of package schemas without having to issue a SET CURRENT PACKAGESET statement each time it switches between SQLJ and JDBC.

For example, assume that the following packages exist, and, using the following list, that you want to invoke the first one that exists on the server: SCHEMA1.PKG1, SCHEMA2.PKG2, SCHEMA3.PKG3, SCHEMA.PKG, and SCHEMA5.PKG5. Assuming the current support for a SET CURRENT PACKAGESET statement in DB2 Universal Database for Linux, UNIX, and Windows (that is, accepting a single schema name), a SET CURRENT PACKAGESET statement would have to be issued before trying to invoke each package to specify the specific schema. For this example, five SET CURRENT PACKAGESET statements would need to be issued. However, using the CURRENT PACKAGE PATH special register, a single SET statement is sufficient. For example:

```
SET CURRENT PACKAGE PATH = SCHEMA1, SCHEMA2, SCHEMA3, SCHEMA, SCHEMA5;
```

Note: In DB2 Universal Database for Linux, UNIX, Windows, you can set the CURRENT PACKAGE PATH special register in the db2cli.ini file, by using the SQLSetConnectAttr API, in the SQLE-CLIENT-INFO structure, and by including the SET CURRENT PACKAGE PATH statement in embedded SQL programs. Only DB2 Universal Database for OS/390 and z/OS, Version 8 or later, supports the SET CURRENT PACKAGE PATH statement. If you issue this statement against a DB2 Universal Database for Linux, UNIX, Windows server or against DB2 Universal Database for AS/00, -30005 is returned.

You can use multiple schemas to maintain several variations of a package. These variations can be a very useful in helping to control changes made in production environments. You can also use different variations of a package to keep a backup version of a package, or a test version of a package (for example, to evaluate the impact of a new index). A previous version of a package is used in the same way as a backup application (load module or executable), specifically, to provide the ability to revert to a previous version.

For example, assume the PROD schema includes the current packages used by the production applications, and the BACKUP schema stores a backup copy of those packages. A new version of the application (and thus the packages) are promoted to production by binding them using the PROD schema. The backup copies of the packages are created by binding the current version of the applications using the backup schema (BACKUP). Then, at runtime, you can use the SET CURRENT PACKAGE PATH statement to specify the order in which the schemas should be checked for the packages. Assume that a backup copy of the application MYAPPL has been bound using the BACKUP schema, and the version of the application currently in production has been bound to the PROD schema creating a package PROD.MYAPPL. To specify that the variation of the package in the PROD schema should be used if it is available (otherwise the variation in the BACKUP schema is used), issue the following SET statement for the special register:

```
SET CURRENT PACKAGE PATH = PROD, BACKUP;
```

If you need to revert to the previous version of the package, the production version of the application can be dropped with the DROP PACKAGE statement, which causes the old version of the application (load module or executable) that was bound using the BACKUP schema to be invoked instead (application path techniques could be used here, specific to each operating system platform).

Note: This example assumes that the only difference between the versions of the package are in the BIND options that were used to create the packages (that is, there are no differences in the executable code).

The application does not use the SET CURRENT PACKAGESET statement to select the schema it wants. Instead, it allows DB2 to pick up the package by checking for it in the schemas listed in the CURRENT PACKAGE PATH special register.

Note: The DB2 Universal Database for OS/390 and z/OS precompile process stores a consistency token in the DBRM (which can be set using the LEVEL option), and during package resolution a check is made to ensure that the consistency token in the program matches the package. Similarly, the DB2 Universal Database for Linux, UNIX, Windows bind process stores a timestamp in the bind file. DB2 Universal Database for Linux, UNIX, Windows also supports a LEVEL option.

Another reason for creating several versions of a package in different schemas could be to cause different BIND options to be in affect. For example, you can use different qualifiers for unqualified name references in the package.

Applications are often written with unqualified table names. This supports multiple tables that have identical table names and structures, but different qualifiers to distinguish different instances. For example, a test system and a production system might have the same objects created in each, but they might have different qualifiers (for example, PROD and TEST). Another example is an application that horizontally partitions data into different tables across different DB2 systems, each with a different qualifier (for example, EAST, WEST, NORTH, SOUTH; COMPANYA, COMPANYB; Y1999, Y2000, Y2001.). With DB2 Universal Database for OS/390 and z/OS, you specify the table qualifier using the QUALIFIER option of the BIND command. When you use the QUALIFIER option, users do not have to maintain multiple programs, each of which specifies the fully qualified names that are required to access unqualified tables. Instead, the correct package can be accessed at runtime by issuing the SET CURRENT PACKAGESET statement from the application, and specifying a single schema name. However, if you use SET CURRENT PACKAGESET, multiple applications will still need to be kept and modified: each one with its own SET CURRENT PACKAGESET statement to access the required package. If you issue a SET CURRENT PACKAGE PATH statement instead, all of the schemas could be listed. At execution time, DB2 could choose the correct package.

Note: DB2 Universal Database for Linux, UNIX, Windows also supports a QUALIFIER bind option. However, the QUALIFIER bind option only affects static SQL or packages that use the DYNAMICRULES option of the BIND command.

Resolution of Unqualified Table Names

You can handle unqualified table names in your application by using one of the following methods:

- For each user, bind the package with different COLLECTION parameters from different authorization identifiers by using the following commands:

  ```
  CONNECT TO db_name USER user_name
  BIND file_name COLLECTION schema_name
  ```

  In the above example, *db_name* is the name of the database, *user_name* is the name of the user, and *file_name* is the name of the application that will be bound. Note that *user_name* and *schema_name* are usually the same value. Then use the SET CURRENT PACKAGESET statement to specify which package to use, and therefore, which qualifiers will be used. The default qualifier is the authorization identifier that is used when binding the package.

- Create views for each user with the same name as the table so the unqualified table names resolve correctly.
- Create an alias for each user to point to the desired table.

Related reference:

- "SET CURRENT PACKAGESET statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*

Additional Considerations when Binding

If your application code page uses a different code page from your database code page, you may need to consider which code page to use when binding.

If your application issues calls to any of the database manager utility APIs, such as IMPORT or EXPORT, you must bind the supplied utility bind files to the database.

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

- The QUERYOPT bind option takes advantage of a specific optimization class when binding.
- The EXPLSNAP bind option stores Explain Snapshot information for eligible SQL statements in the Explain tables.
- The FUNCPATH bind option properly resolves user-defined distinct types and user-defined functions in static SQL.

If the bind process starts but never returns, it may be that other applications connected to the database hold locks that you require. In this case, ensure that no applications are connected to the database. If they are, disconnect all applications on the server and the bind process will continue.

If your application will access a server using DB2 Connect, you can use the `BIND` options available for that server.

Bind files are not backward compatible with previous versions of DB2 Universal Database. In mixed-level environments, DB2® can only use the functions available to the lowest level of the database environment. For example, if a V8 client connects to a V7.2 server, the client will only be able to use V7.2 functions. As bind files express the functionality of the database, they are subject to the mixed-level restriction.

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

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

Related concepts:

- "Binding utilities to the database" in the *Administration Guide: Implementation*
- "Character conversion between different code pages" on page 609
- "Character Substitutions During Code Page Conversions" on page 610
- "Code Page Conversion Expansion Factor" on page 611

Related reference:

- "BIND Command" in the *Command Reference*

Advantages of Deferred Binding

Precompiling with binding enabled allows an application to access only the database used during the precompile process. Precompiling with binding deferred, however, allows an application to access many databases, because you can bind the BIND file against each one. This method of application development is inherently more flexible in that applications are precompiled only once, but the application can be bound to a database at any time.

Using the BIND API during execution allows an application to bind itself, perhaps as part of an installation procedure or before an associated module is executed. For example, an application can perform several tasks, only one of which requires the use of SQL statements. You can design the application to bind itself to a database only when the application calls the task requiring SQL statements, and only if an associated package does not already exist.

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

Related reference:

- "sqlabndx - Bind" in the *Administrative API Reference*

Bind File Contents

With the DB2® Bind File Description (`db2bfd`) utility, you can easily display the contents of a bind file to examine and verify the SQL statements within it, as well as display the precompile options used to create the bind file. This may be useful in problem determination related to your application's bind file.

Related reference:

- "db2bfd - Bind File Description Tool Command" in the *Command Reference*

Application, Bind File, and Package Relationships

A package is an object stored in the database that includes information needed to execute specific SQL statements in a single source file. A database application uses one package for every precompiled source file used to build the application. Each package is a separate entity, and has no relationship to any other packages used by the same or other applications. Packages are created by running the precompiler against a source file with binding enabled, or by running the binder at a later time with one or more bind files.

Database applications use packages for some of the same reasons that applications are compiled: improved performance and compactness. By precompiling an SQL statement, the statement is compiled into the package when the application is built, instead of at run time. Each statement is parsed, and a more efficiently interpreted operand string is stored in the package. At run time, the code generated by the precompiler calls run-time services database manager APIs with any variable information required for input or output data, and the information stored in the package is executed.

The advantages of precompilation apply only to static SQL statements. SQL statements that are executed dynamically (using PREPARE and EXECUTE or EXECUTE IMMEDIATE) are not precompiled; therefore, they must go through the entire set of processing steps at run time.

Note: Do not assume that a static version of an SQL statement automatically executes faster than the same statement processed dynamically. In some cases, static SQL is faster because of the overhead required to prepare the dynamic statement. In other cases, the same statement prepared dynamically executes faster, because the optimizer can make use of current database statistics, rather than the database statistics available at an earlier bind time. Note that if your transaction takes less than a couple of seconds to complete, static SQL will generally be faster. To choose which method to use, you should prototype both forms of binding.

Related concepts:

• "Dynamic SQL Versus Static SQL" on page 104

Precompiler-Generated Timestamps

When generating a package or a bind file, the precompiler generates a timestamp. The timestamp is stored in the bind file or package and in the modified source file. The timestamp is also known as the *consistency token*.

When an application is precompiled with binding enabled, the package and modified source file are generated with timestamps that match. If multiple versions of a package exist (by using the PRECOMPILE VERSION option), each version will have with it an associated timestamp. When the application is run, the package name, creator and timestamp are sent to the database manager, which checks for a package whose name, creator and timestamp match that sent by the application. If such a match does not exist, one of the two following SQL error codes is returned to the application:

- SQL0818N (timestamp conflict). This error is returned if a single package is found that matches the name and creator (but not the consistency token), and the package has a version of "" (an empty string)
- SQL0805N (package not found). This error is returned in all other situations.

Remember that when you bind an application to a database, the first eight characters of the application name are used as the package name *unless you override the default by using the PACKAGE USING option on the PREP command*. As well the version ID will be "" (an empty string) unless it is specified by the VERSION option of the PREP command. This means that if you precompile and bind two programs using the same name without changing the version ID, the second package will replace the package of the first. When you run the first program, you will get a timestamp or a package not found error because the timestamp for the modified source file no longer matches that of the package in the database. The package not found error can also result from the use of the ACTION REPLACE REPLVER precompile or bind option as in the following example:

1. Precompile and bind the package SCHEMA1.PKG specifying VERSION VER1. Then generate the associated application A1.
2. Precompile and bind the package SCHEMA1.PKG, specifying VERSION VER2 ACTION REPLACE REPLVER VER1. Then generate the associated application A2.

The second precompile and bind generates a package SCHEMA1.PKG that has a VERSION of VER2, and the specification of ACTION REPLACE REPLVER VER1 removes the SCHEMA1.PKG package that had a VERSION of VER1.

An attempt to run the first application will result in a package mismatch and will fail.

A similar symptom will occur in the following example:

1. Precompile and bind teh package SCHEMA1.PKG, specifying VERSION VER1. Then generate the associated application A1
2. Precompile and bind the package SCHEMA1.PKG, specifying VERSION VER2. Then generate the associated application A2

   At this point it is possible to run both applications A1 and A2, which will execute from packages SCHEMA1.PKG versions VER1 and VER2 respectively. If, for example, the first package is dropped (using the DROP PACKAGE SCHEMA1.PKG VERSION VER1 SQL statement), an attempt to run the application A1 will fail with a package not found error.

When a source file is precompiled but a respective package is not created, a bind file and modified source file are generated with matching timestamps. To run the application, the bind file is bound in a separate BIND step to create a package and the modified source file is compiled and linked. For an application that requires multiple source modules, the binding process must be done for each bind file.

In this deferred binding scenario, the application and package timestamps match because the bind file contains the same timestamp as the one that was stored in the modified source file during precompilation.

Related concepts:

• "Package Creation Using the BIND Command" on page 64

Package Rebinding

Rebinding is the process of recreating a package for an application program that was previously bound. You must rebind packages if they have been marked invalid or inoperative. In some situations, however, you may want to rebind packages that are valid. For example, you may want to take advantage of a newly created index, or make use of updated statistics after executing the RUNSTATS command.

Packages can be dependent on certain types of database objects such as tables, views, aliases, indexes, triggers, referential constraints and table check constraints. If a package is dependent on a database object (such as a table, view, trigger, and so on), and that object is dropped, the package is placed into an *invalid* state. If the object that is dropped is a UDF, the package is placed into an *inoperative* state.

Invalid packages are implicitly (or automatically) rebound by the database manager when they are executed. Inoperative packages must be explicitly rebound by executing either the `BIND` command or the `REBIND` command. Note that implicit rebinding can cause unexpected errors if the implicit rebind fails. That is, the implicit rebind error is returned on the statement being executed, which may not be the statement that is actually in error. If an attempt is made to execute an inoperative package, an error occurs. You may decide to explicitly rebind invalid packages rather than have the system automatically rebind them. This enables you to control when the rebinding occurs.

The choice of which command to use to explicitly rebind a package depends on the circumstances. You must use the `BIND` command to rebind a package for a program which has been modified to include more, fewer, or changed SQL statements. You must also use the `BIND` command if you need to change any bind options from the values with which the package was originally bound. In all other cases, use either the `BIND` or `REBIND` command. You should use `REBIND` whenever your situation does not specifically require the use of `BIND`, as the performance of `REBIND` is significantly better than that of `BIND`.

When multiple versions of the same package name coexist in the catalog, only one version at a time can be rebound.

Related concepts:

- "Statement dependencies when changing objects" in the *Administration Guide: Implementation*

Related reference:

- "BIND Command" in the *Command Reference*
- "REBIND Command" in the *Command Reference*

Chapter 4. Writing Static SQL Programs

Characteristics and Reasons for Using Static SQL

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as *static* SQL. This is in contrast to *dynamic* SQL statements whose syntax is not known until run time.

Note: Static SQL is not supported in interpreted languages, such as REXX.

The structure of an SQL statement must be completely specified for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at run time are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled.

When a static SQL statement is prepared, an executable form of the statement is created and stored in the package in the database. The executable form can be constructed either at precompile time, or at a later bind time. In either case, preparation occurs *before* run time. The authorization of the person binding the application is used, and optimization is based upon database statistics and configuration parameters that may not be current when the application runs.

Advantages of Static SQL

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time services API calls that the host language compiler can process.

Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or to a range of values.

Static SQL statements are *persistent*, meaning that the statements last for as long as the package exists.

Dynamic SQL statements are cached until they are either invalidated, freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2® SQL compiler whenever a cached statement becomes invalid.

The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at run time, while dynamic SQL must be explicitly compiled at run time (for example, by using the PREPARE statement). Because DB2 caches dynamic SQL statements, the statements do not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.

There can be performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically because the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

Note: The performance of static SQL depends on the statistics of the database the last time the application was bound. However, if these statistics change, the performance of equivalent dynamic SQL can be very different. If, for example, an index is added to a database at a later time, an application using static SQL cannot take advantage of the index unless it is rebound to the database. In addition, if you are using host variables in a static SQL statement, the optimizer will not be able to take advantage of any distribution statistics for the table.

Related reference:

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

Example Static SQL Program

This sample program shows examples of static SQL statements and database manager API calls in the C/C++, Java™, and COBOL languages.

The sample in C/C++ and Java queries the **org** table in the sample database to find the department name and department number of the department that is located in New York, then places the department name and department number into host variables.

The sample in COBOL queries the **employee** table in the sample database for the first name of the employee whose last name is Johnson, then place the first name into a host variable.

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

- C/C++ (**tbread**)

```
SELECT deptnumb, deptname INTO :deptnumb, :deptname
FROM org
WHERE location = 'New York'
```

  This query is in the `TbRowSubselect()` function of the sample. For more information, see the related samples below.

- Java (TbRead.sqlj)

```
#sql cur2 = {SELECT deptnumb, deptname
 FROM org
 WHERE location = 'New York'};
// fetch the cursor
#sql {FETCH :cur2 INTO :deptnumb, :deptname};
```

  This query is in the `rowSubselect()` function of the **TbRead.sqlj** sample. For more information, see the related samples below.

- COBOL (**static.sqb**)

  The sample **static** contains examples of static SQL statements and database manager API calls in the COBOL language. The SELECT INTO statement selects one row of data from tables in a database, and the values in this row are assigned to host variables specified in the statement. For example, the following statement delivers the first name of the employee with the last name JOHNSON into the host variable `firstname`:

```
SELECT FIRSTNME
 INTO :firstname
 FROM EMPLOYEE
 WHERE LASTNAME = 'JOHNSON'
```

Related concepts:

- "Data Retrieval in Static SQL Programs" on page 78
- "Error Message Retrieval in an Application" on page 102

Related tasks:

- "Declaring Host Variables in Static SQL Programs" on page 80
- "Selecting Multiple Rows Using a Cursor" on page 87
- "Setting up the sample database" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "SELECT INTO statement" in the *SQL Reference, Volume 2*

Related samples:

- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"

- "tbinfo.out -- HOW TO GET INFORMATION AT THE TABLE LEVEL (C)"
- "tbread.out -- HOW TO READ TABLES (C)"
- "tbread.sqc -- How to read tables (C)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "tbinfo.sqC -- How to get information at the table level (C++)"
- "tbread.out -- HOW TO READ TABLES (C++)"
- "tbread.sqC -- How to read tables (C++)"
- "static.sqb -- Get table data using static SQL statement (IBM COBOL)"
- "static.sqb -- Get table data using static SQL statement (MF COBOL)"
- "TbRead.out -- HOW TO READ TABLE DATA. Connect to 'sample' database using JDBC type 2 driver (JDBC)"
- "TbRead.sqlj -- How to read table data (SQLj)"

Data Retrieval in Static SQL Programs

One of the most common tasks of an SQL application program is to retrieve data. This task is done using the *select-statement*, which is a form of query that searches for rows of tables in the database that meet specified search conditions. If such rows exist, the data is retrieved and put into specified variables in the host program, where it can be used for whatever it was designed to do.

After you have written a select-statement, you code the SQL statements that define how information will be passed to your application.

You can think of the result of a select-statement as being a table having rows and columns, much like a table in the database. If only one row is returned, you can deliver the results directly into host variables specified by the SELECT INTO statement.

If more than one row is returned, you must use a *cursor* to fetch them one at a time. A cursor is a named control structure used by an application program to point to a specific row within an ordered set of rows.

Related concepts:
- "Host Variables in Static SQL" on page 79
- "Example of a Cursor in a Static SQL Program" on page 90

Related tasks:
- "Declaring Host Variables in Static SQL Programs" on page 80
- "Referencing Host Variables in Static SQL Programs" on page 81
- "Including Indicator Variables in Static SQL Programs" on page 82
- "Selecting Multiple Rows Using a Cursor" on page 87
- "Declaring and Using Cursors in Static SQL Programs" on page 88

Effects of REOPT on static SQL

The bind option REOPT can make static SQL statements containing host variables or special registers behave like incremental-bind statements. This means that these statements get compiled at the time of EXECUTE or OPEN instead of at bind time. During this compilation, the access plan is chosen, based on the real values of these variables.

With REOPT ONCE, the access plan is cached after the first OPEN or EXECUTE request and is used for subsequent execution of this statement. With REOPT ALWAYS, the access plan is regenerated for every OPEN and EXECUTE request, and the current set of host variable, parameter marker, and special register values is used to create this plan.

Host Variables in Static SQL Programs

The sections that follow describe how to use host variables in static SQL programs.

Host Variables in Static SQL

Host variables are variables referenced by embedded SQL statements. They transmit data between the database manager and an application program. When you use a host variable in an SQL statement, you must prefix its name with a colon, (**:**). When you use a host variable in a host language statement, omit the colon.

Host variables are declared in compiled host languages, and are delimited by BEGIN DECLARE SECTION and END DECLARE SECTION statements. These statements enable the precompiler to find the declarations.

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

Host variables are declared using a subset of the host language.

The following rules apply to host variable declaration sections:

- All host variables must be declared in the source file before they are referenced, except for host variables referring to SQLDA structures.
- Multiple declare sections may be used in one source file.
- The precompiler is unaware of host language variable scoping rules.

  With respect to SQL statements, all host variables have a global scope regardless of where they are actually declared in a single source file. Therefore, host variable names must be unique within a source file.

  This does not mean that the DB2® precompiler changes the scope of host variables to global so that they can be accessed outside the scope in which they are defined. Consider the following example:

```
foo1(){
  .
  .
  .
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
x=10;
  .
  .
  .
}


foo2(){
  .
  .
  .
  y=x;
```

```
  .
  .
  .
}
```

Depending on the language, the above example will either fail to compile because variable `x` is not declared in function `foo2()`, or the value of `x` would not be set to 10 in `foo2()`. To avoid this problem, you must either declare `x` as a global variable, or pass `x` as a parameter to function `foo2()` as follows:

```
foo1(){
.
.
.
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
  x=10;
  foo2(x);
.
.
.
}


foo2(int x){
.
.
.
  y=x;
.
.
.
}
```

Related concepts:

- "Host Variables in C and C++" on page 137
- "Host Variables in COBOL" on page 180
- "Host Variables in FORTRAN" on page 200
- "Host Variables in REXX" on page 497

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Declaring Host Variables in Static SQL Programs" on page 80
- "Referencing Host Variables in Static SQL Programs" on page 81

Declaring Host Variables in Static SQL Programs

Declare host variables for your program so that they can be used to transmit data between the database manager and the application.

Procedure:

Declare the host variables using the syntax for the host language that you are using. The following table provides examples.

Table 4. Host Variable Declarations by Host Language

Language	Example Source Code

C/C++	`EXEC SQL BEGIN DECLARE SECTION;`
	`  short  dept=38, age=26;`
	`  double  salary;`
	`  char  CH;`
	`  char  name1[9], NAME2[9];`
	`  /* C comment */`
	`  short  nul_ind;`
	`EXEC SQL END DECLARE SECTION;`

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

COBOL	`EXEC SQL BEGIN DECLARE SECTION END-EXEC.`
	`  01 age  PIC S9(4) COMP-5 VALUE 26.`
	`  01 DEPT  PIC S9(9) COMP-5 VALUE 38.`
	`  01 salary  PIC S9(6)V9(3) COMP-3.`
	`  01 CH  PIC X(1).`
	`  01 name1  PIC X(8).`
	`  01 NAME2  PIC X(8).`
	`*  COBOL comment`
	`  01 nul-ind  PIC S9(4) COMP-5.`
	`EXEC SQL END DECLARE SECTION END-EXEC.`

FORTRAN	`  EXEC SQL BEGIN DECLARE SECTION`
	`  integer*2  age  /26/`
	`  integer*4  dept  /38/`
	`  real*8  salary`
	`  character  ch`
	`  character*8  name1,NAME2`
	`C  FORTRAN comment`
	`  integer*2  nul_ind`
	`  EXEC SQL END DECLARE SECTION`

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Referencing Host Variables in Static SQL Programs" on page 81

Referencing Host Variables in Static SQL Programs

After declaring the host variable, you can reference it in the application program. When you use a host variable in an SQL statement, prefix its name with a colon (:). If you use a host variable in a host language statement, omit the colon.

Procedure:

Reference the host variables using the syntax for the host language that you are using. The following table provides examples.

Table 5. Host Variable Refrerences by Host Language

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

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Declaring Host Variables in Static SQL Programs" on page 80

Indicator Variables in Static SQL Programs

The sections that follow describe how to use indicator variables in static SQL programs.

Including Indicator Variables in Static SQL Programs

Applications written in languages other than Java must prepare for receiving null values by associating an *indicator variable* with any host variable that can receive a null. Java applications compare the value of the host variable with Java `null` to determine whether the received value is null. An indicator variable is shared by both the database manager and the host application; therefore, the indicator variable must be declared in the application as a host variable. This host variable corresponds to the SQL data type SMALLINT.

An indicator variable is placed in an SQL statement immediately after the host variable, and is prefixed with a colon. A space can separate the indicator variable from the host variable, but is not required. However, do not put a comma between the host variable and the indicator variable. You can also specify an indicator variable by using the optional INDICATOR keyword, which you place between the host variable and its indicator.

Procedure:

Use the INDICATOR keyword to write indicator variables. The following table provides examples for the supported host languages:

Table 6. Indicator Variables by Host Language

Language	Example Source Code

C/C++	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;`
	`if ( cmind < 0 )`
	`  printf( "Commission is NULL\n" );`

JAVA (SQLJ)	`#SQL { FETCH :c1 INTO :cm };`
	`if ( cm == null )`
	`  System.out.println( "Commission is NULL\n" );`

Table 6. Indicator Variables by Host Language (continued)

Language	Example Source Code

COBOL	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC`
	`IF cmind LESS THAN 0`
	`  DISPLAY 'Commission is NULL'`

FORTRAN	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind`
	`IF ( cmind .LT. 0 ) THEN`
	`  WRITE(*,*) 'Commission is NULL'`
	`ENDIF`

In the preceding examples, *cmind* is examined for a negative value. If the value is not negative, the application can use the returned value of *cm*. If the value is negative, the fetched value is NULL and *cm* should not be used. The database manager does not change the value of the host variable in this case.

Note: If the database configuration parameter *dft_sqlmathwarn* is set to 'YES', the value of *cmind* may be -2. This value indicates a NULL that was either caused by evaluating an expression with an arithmetic error, or by an overflow while attempting to convert the numeric result value to the host variable.

If the data type can handle NULLs, the application must provide a NULL indicator. Otherwise, an error may occur. If a NULL indicator is not used, an SQLCODE -305 (SQLSTATE 22002) is returned.

If the SQLCA structure indicates a truncation warning, the indicator variables can be examined for truncation. If an indicator variable has a positive value, a truncation occurred.

- If the seconds portion of a TIME data type is truncated, the indicator value contains the seconds portion of the truncated data.
- For all other string data types, except large objects (LOB), the indicator value represents the actual length of the data returned. User-defined distinct types (UDT) are handled in the same way as their base type.

When processing INSERT or UPDATE statements, the database manager checks the indicator variable if one exists. If the indicator variable is negative, the database manager sets the target column value to NULL if NULLs are allowed.

If the indicator variable is zero or positive, the database manager uses the value of the associated host variable.

The SQLWARN1 field in the SQLCA structure may contain an `X` or `W` if the value of a string column is truncated when it is assigned to a host variable. The field contains an `N` if a null terminator is truncated.

A value of `X` is returned by the database manager only if all of the following conditions are met:

- A mixed code page connection exists where conversion of character string data from the database code page to the application code page involves a change in the length of the data.
- A cursor is blocked.
- An indicator variable is provided by your application.

The value returned in the indicator variable will be the length of the resultant character string in the application's code page.

In all other cases involving data truncation (as opposed to NULL terminator truncation), the database manager returns a `W`. In this case, the database manager returns a value in the indicator variable to the application that is the length of the resultant character string in the code page of the select list item (either the application code page, the database code page, or nothing).

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Declaring Host Variables in Static SQL Programs" on page 80
- "Referencing Host Variables in Static SQL Programs" on page 81

Related reference:

- "Data Types for Indicator Variables in Static SQL Programs" on page 84

Data Types for Indicator Variables in Static SQL Programs

Each column of every DB2 table is given an *SQL data type* when the column is created. For information about how these types are assigned to columns, see the CREATE TABLE statement. The database manager supports the following column data types:

SMALLINT
16-bit signed integer.

INTEGER
32-bit signed integer. **INT** can be used as a synonym for this type.

BIGINT
64-bit signed integer.

DOUBLE
Double-precision floating point. **DOUBLE PRECISION** and **FLOAT(*n*)** (where *n* is greater than 24) are synonyms for this type.

REAL Single-precision floating point. **FLOAT(*n*)** (where *n* is less than 24) is a synonym for this type.

DECIMAL
Packed decimal. **DEC**, **NUMERIC**, and **NUM** are synonyms for this type.

CHAR
Fixed-length character string of length 1 byte to 254 bytes. **CHARACTER** can be used as a synonym for this type.

VARCHAR
Variable-length character string of length 1 byte to 32 672 bytes. **CHARACTER VARYING** and **CHAR VARYING** are synonyms for this type.

LONG VARCHAR
Long variable-length character string of length 1 byte to 32 700 bytes.

CLOB Large object variable-length character string of length 1 byte to 2 gigabytes.

BLOB Large object variable-length binary string of length 1 byte to 2 gigabytes.

DATE Character string of length 10 representing a date.

TIME Character string of length 8 representing a time.

TIMESTAMP
Character string of length 26 representing a timestamp.

The following data types are supported only in double-byte character set (DBCS) and Extended UNIX Code (EUC) character set environments:

GRAPHIC
Fixed-length graphic string of length 1 to 127 double-byte characters.

VARGRAPHIC
Variable-length graphic string of length 1 to 16 336 double-byte characters.

LONG VARGRAPHIC
Long variable-length graphic string of length 1 to 16 350 double-byte characters.

DBCLOB
Large object variable-length graphic string of length 1 to 1 073 741 823 double-byte characters.

Notes:

1. Every supported data type can have the NOT NULL attribute. This is treated as another type.
2. The above set of data types can be extended by defining user-defined distinct types (UDT). UDTs are separate data types that use the representation of one of the built-in SQL types.

Supported host languages have data types that correspond to the majority of the database manager data types. Only these host language data types can be used in host variable declarations. When the precompiler finds a host variable declaration, it determines the appropriate SQL data type value. The database manager uses this value to convert the data exchanged between itself and the application.

As the application programmer, it is important for you to understand how the database manager handles comparisons and assignments between different data types. Simply put, data types must be compatible with each other during assignment and comparison operations, whether the database manager is working with two SQL column data types, two host-language data types, or one of each.

The *general* rule for data type compatibility is that all supported host-language numeric data types are comparable and assignable with all database manager numeric data types, and all host-language character types are compatible with all database manager character types; numeric types are incompatible with character types. However, there are also some exceptions to this general rule, depending on host language idiosyncrasies and limitations imposed when working with large objects.

Within SQL statements, DB2 provides conversions between compatible data types. For example, in the following SELECT statement, SALARY and BONUS are DECIMAL columns; however, each employee's total compensation is returned as DOUBLE data:

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

Note that the execution of the above statement includes conversion between DECIMAL and DOUBLE data types.

To make the query results more readable on your screen, you could use the following SELECT statement:

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

To convert data within your application, contact your compiler vendor for additional routines, classes, built-in types, or APIs that support this conversion.

If your application code page is not the same as your database code page, character data types may also be subject to character conversion.

Related concepts:

- "Data conversion considerations" in the *Application Development Guide: Programming Server Applications*
- "Character conversion between different code pages" on page 609

Related reference:

- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in C and C++" on page 162
- "Supported SQL Data Types in COBOL" on page 190
- "Supported SQL Data Types in FORTRAN" on page 206
- "Supported SQL Data Types in REXX" on page 502
- "Java, JDBC, and SQL data types" on page 365

Example of an Indicator Variable in a Static SQL Program

Following are examples of how to use indicator variables C/C++ programs that use have static SQL:

- Example 1

  The following example show the implementation of indicator variables on data columns that are nullable. In this example, the column FIRSTNAME is not nullable, but the column WORKDEPT can contain a null value.

```
EXEC SQL BEGIN DECLARE SECTION;
  char wd[3];
  short wd_ind;
  char firstname[13];
EXEC SQL END DECLARE SECTION;

  /* connect to sample database */

EXEC SQL SELECT FIRSTNME, WORKDEPT
  INTO :firstname, :wd:wdind
  FROM EMPLOYEE
  WHERE LASTNAME = 'JOHNSON';
```

  Because the column WORKDEPT can have a null value, an indicator variable must be declared as a host variable before being used.

- Example 2 (**dtlob**)

  The sample **dtlob** has a function called `BlobFileUse()`. The function `BlobFileUse()` contains a query that reads BLOB data in a file using a SELECT INTO statement:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS BLOB_FILE blobFilePhoto;
  char photoFormat[10];
  char empno[7];
  short lobind;
```

```
EXEC SQL END DECLARE SECTION;

  /* Connect to the sample database */

SELECT picture INTO :blobFilePhoto:lobind
  FROM emp_photo
  WHERE photo_format = :photoFormat AND empno = '000130'
```

Because the column BLOBFILEPHOTO can have a null value, an indicator variable LOBIND must be declared as a host variable before being used. The sample **dtlob** shows how to work with LOBs. See the samples for more information about using LOBs.

Related concepts:

- "Example Static SQL Program" on page 76

Related tasks:

- "Including Indicator Variables in Static SQL Programs" on page 82

Related reference:

- "Data Types for Indicator Variables in Static SQL Programs" on page 84

Related samples:

- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"

Selecting Multiple Rows Using a Cursor

The sections that follow describe how to select rows using a cursor. The sample programs that show how to declare a cursor, open the cursor, fetch rows from the table, and close the cursor are also briefly described.

Selecting Multiple Rows Using a Cursor

To allow an application to retrieve a set of rows, SQL uses a mechanism called a *cursor*.

To help understand the concept of a cursor, assume that the database manager builds a *result table* to hold all the rows retrieved by executing a SELECT statement. A cursor makes rows from the result table available to an application by identifying or pointing to a *current row* of this table. When a cursor is used, an application can retrieve each row sequentially from the result table until an end of data condition, that is, the NOT FOUND condition, SQLCODE +100 (SQLSTATE 02000) is reached. The set of rows obtained as a result of executing the SELECT statement can consist of zero, one, or more rows, depending on the number of rows that satisfy the search condition.

Procedure:

The steps involved in processing a cursor are as follows:

1. Specify the cursor using a DECLARE CURSOR statement.
2. Perform the query and build the result table using the OPEN statement.

3. Retrieve rows one at a time using the FETCH statement.
4. Process rows with the DELETE or UPDATE statements (if required).
5. Terminate the cursor using the CLOSE statement.

An application can use several cursors concurrently. Each cursor requires its own set of DECLARE CURSOR, OPEN, CLOSE, and FETCH statements.

Related concepts:

- "Example of a Cursor in a Static SQL Program" on page 90

Declaring and Using Cursors in Static SQL Programs

Use the DECLARE CURSOR statement to define and name the cursor, and to identify the set of rows to be retrieved using a SELECT statement.

The application assigns a name for the cursor. This name is referred to in subsequent OPEN, FETCH, and CLOSE statements. The query is any valid select statement.

Restrictions:

The placement of the DECLARE statement is arbitrary, but it must be placed above the first use of the cursor.

Procedure:

Use the DECLARE statement to define the cursor. The following table provides examples for the supported host languages:

Table 7. Cursor Declarations by Host Language

Language	Example Source Code

C/C++	`EXEC SQL DECLARE C1 CURSOR FOR`
	`  SELECT PNAME, DEPT FROM STAFF`
	`  WHERE JOB=:host_var;`

JAVA (SQLJ)	`#sql iterator cursor1(host_var data type);`
	`#sql cursor1 = { SELECT PNAME, DEPT FROM STAFF`
	`  WHERE JOB=:host_var };`

COBOL	`EXEC SQL DECLARE C1 CURSOR FOR`
	`  SELECT NAME, DEPT FROM STAFF`
	`  WHERE JOB=:host-var END-EXEC.`

FORTRAN	`EXEC SQL DECLARE C1 CURSOR FOR`
	`+  SELECT NAME, DEPT FROM STAFF`
	`+  WHERE JOB=:host_var`

Related concepts:

- "Cursor Types and Unit of Work Considerations" on page 89

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 87

Related reference:

- "Cursor Types" on page 92

Cursor Types and Unit of Work Considerations

The actions of a COMMIT or ROLLBACK operation vary for cursors, depending on how the cursors are declared:

Read-only cursors

If a cursor is determined to be read only and uses a repeatable read isolation level, repeatable read locks are still gathered and maintained on system tables needed by the unit of work. Therefore, it is important for applications to periodically issue COMMIT statements, even for read only cursors.

WITH HOLD option

If an application completes a unit of work by issuing a COMMIT statement, *all open cursors*, except those declared using the WITH HOLD option, are automatically closed by the database manager.

A cursor that is declared WITH HOLD maintains the resources it accesses across multiple units of work. The exact effect of declaring a cursor WITH HOLD depends on how the unit of work ends:

- If the unit of work ends with a COMMIT statement, open cursors defined WITH HOLD remain OPEN. The cursor is positioned before the next logical row of the result table. In addition, prepared statements referencing OPEN cursors defined WITH HOLD are retained. Only FETCH and CLOSE requests associated with a particular cursor are valid immediately following the COMMIT. UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF statements are valid only for rows fetched within the same unit of work.

  Note: If a package is rebound during a unit of work, all held cursors are closed.
- If the unit of work ends with a ROLLBACK statement, all open cursors are closed, all locks acquired during the unit of work are released, and all prepared statements that are dependent on work done in that unit are dropped.

For example, suppose that the TEMPL table contains 1 000 entries. You want to update the salary column for all employees, and you expect to issue a COMMIT statement every time you update 100 rows.

1. Declare the cursor using the WITH HOLD option:

```
EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
  FROM TEMPL FOR UPDATE OF SALARY
```

2. Open the cursor and fetch data from the result table one row at a time:

```
EXEC SQL OPEN EMPLUPDT
  .
  .
  .

EXEC SQL FETCH EMPLUPDT
  INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
```

3. When you want to update or delete a row, use an UPDATE or DELETE statement using the WHERE CURRENT OF option. For example, to update the current row, your program can issue:

```
EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
  WHERE CURRENT OF EMPLUPDT
```

4. After a COMMIT is issued, you must issue a FETCH before you can update another row.

You should include code in your application to detect and handle an SQLCODE -501 (SQLSTATE 24501), which can be returned on a FETCH or CLOSE statement if your application either:

- Uses cursors declared WITH HOLD
- Executes more than one unit of work and leaves a WITH HOLD cursor open across the unit of work boundary (COMMIT WORK).

If an application invalidates its package by dropping a table on which it is dependent, the package gets rebound dynamically. If this is the case, an SQLCODE -501 (SQLSTATE 24501) is returned for a FETCH or CLOSE statement because the database manager closes the cursor. The way to handle an SQLCODE -501 (SQLSTATE 24501) in this situation depends on whether you want to fetch rows from the cursor:

- If you want to fetch rows from the cursor, open the cursor, then run the FETCH statement. Note, however, that the OPEN statement repositions the cursor to the start. The previous position held at the COMMIT WORK statement is lost.
- If you do not want to fetch rows from the cursor, do not issue any more SQL requests against the cursor.

WITH RELEASE option

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

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 87
- "Declaring and Using Cursors in Static SQL Programs" on page 88

Example of a Cursor in a Static SQL Program

The samples **tut_read.sqc** in C, **tut_read.sqC/sqx** in C++, **TutRead.sqlj** in SQLJ, and **cursor.sqb** in COBOL show how to declare a cursor, open the cursor, fetch rows from the table, and close the cursor.

Because REXX does not support static SQL, a sample is not provided.

- C/C++

  The sample tut_read shows a basic select from a table using a cursor. For example:

```
 /* declare cursor */
  EXEC SQL DECLARE c1 CURSOR FOR
    SELECT deptnumb, deptname FROM org WHERE deptnumb < 40;

/* open cursor */
  EXEC SQL OPEN c1;

/* fetch cursor */
  EXEC SQL FETCH c1 INTO :deptnumb, :deptname;

while (sqlca.sqlcode != 100)
  {
    printf("    %8d %-14s\n", deptnumb, deptname);
```

```
  EXEC SQL FETCH c1 INTO :deptnumb, :deptname;
}

/* close cursor */
EXEC SQL CLOSE c1;
```

- Java™

  The sample **TutRead** shows how to read table data with a simple select using a cursor. For example:

```
// cursor definition
  #sql iterator TutRead_Cursor(int, String);

  // declare cursor
  TutRead_Cursor cur2;
  #sql cur2 = {SELECT deptnumb, deptname FROM org WHERE deptnumb < 40};

  // fetch cursor
  #sql {FETCH :cur2 INTO :deptnumb, :deptname};

  // retrieve and display the result from the SELECT statement
  while (!cur2.endFetch())
  {
    System.out.println(deptnumb + ", " + deptname);
    #sql {FETCH :cur2 INTO :deptnumb, :deptname};
  }

  // close cursor
  cur2.close();
```

- COBOL

  The sample **cursor** shows an example on how to retrieve table data using a cursor with Static SQL statement. For example:

```
* Declare a cursor
     EXEC SQL DECLARE c1 CURSOR FOR
              SELECT name, dept FROM staff
              WHERE job='Mgr' END-EXEC.

* Open the cursor
     EXEC SQL OPEN c1 END-EXEC.

* Fetch rows from the 'staff' table
     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

* Close the cursor
     EXEC SQL CLOSE c1 END-EXEC.
     move "CLOSE CURSOR" to errloc.
```

Related concepts:

- "Cursor Types and Unit of Work Considerations" on page 89
- "Error Message Retrieval in an Application" on page 102

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 87
- "Declaring and Using Cursors in Static SQL Programs" on page 88

Related reference:

- "Cursor Types" on page 92

Related samples:

- "cursor.sqb -- How to update table data with cursor statically (IBM COBOL)"
- "tut_read.out -- HOW TO READ TABLES (C)"

- "tut_read.sqc -- How to read tables (C)"
- "tut_read.out -- HOW TO READ TABLES (C++)"
- "tut_read.sqC -- How to read tables (C++)"
- "TutRead.out -- HOW TO READ TABLE DATA. Connect to 'sample' database using JDBC type 2 driver (SQLJ)"
- "TutRead.sqlj -- Read data in a table (SQLj)"

Manipulating Retrieved Data

The sections that follow describe how to update and delete retrieved data. The sample programs that show how to manipulate data are also briefly described.

Updating and Deleting Retrieved Data in Static SQL Programs

It is possible to update and delete the row referenced by a cursor. For a row to be updatable, the query corresponding to the cursor must not be read-only.

Procedure:

To update with a cursor, use the WHERE CURRENT OF clause in an UPDATE statement. Use the FOR UPDATE clause to tell the system that you want to update some columns of the result table. You can specify a column in the FOR UPDATE without it being in the fullselect; therefore, you can update columns that are not explicitly retrieved by the cursor. If the FOR UPDATE clause is specified without column names, all columns of the table or view identified in the first FROM clause of the outer fullselect are considered to be updatable. Do not name more columns than you need in the FOR UPDATE clause. In some cases, naming extra columns in the FOR UPDATE clause can cause DB2 to be less efficient in accessing the data.

Deletion with a cursor is done using the WHERE CURRENT OF clause in a DELETE statement. In general, the FOR UPDATE clause is not required for deletion of the current row of a cursor. The only exception occurs when using dynamic SQL for either the SELECT statement or the DELETE statement in an application that has been precompiled with LANGLEVEL set to SAA1 and bound with BLOCKING ALL. In this case, a FOR UPDATE clause is necessary in the SELECT statement.

The DELETE statement causes the row being referenced by the cursor to be deleted. The deletion leaves the cursor positioned before the *next* row, and a FETCH statement must be issued before additional WHERE CURRENT OF operations may be performed against the cursor.

Related reference:

- "PRECOMPILE Command" in the *Command Reference*
- "SQL queries" in the *SQL Reference, Volume 1*

Cursor Types

Cursors fall into three categories:

Read only
: The rows in the cursor can only be read, not updated. Read-only cursors are used when an application will only read data, not modify it. A cursor is considered read only if it is based on a read-only select-statement. See

the description of how to update and retrieve data for the rules for select-statements that define non-updatable result tables.

There can be performance advantages for read-only cursors.

Updatable
The rows in the cursor can be updated. Updatable cursors are used when an application modifies data as the rows in the cursor are fetched. The specified query can only refer to one table or view. The query must also include the FOR UPDATE clause, naming each column that will be updated (unless the LANGLEVEL MIA precompile option is used).

Ambiguous
The cursor cannot be determined to be updatable or read only from its definition or context. This situation can happen when a dynamic SQL statement is encountered that could be used to change a cursor that would otherwise be considered read-only.

An ambiguous cursor is treated as read only if the BLOCKING ALL option is specified when precompiling or binding. Otherwise, the cursor is considered updatable.

Note: Cursors processed dynamically are always ambiguous.

Related concepts:

- "Supported Cursor Modes for the IBM OLE DB Provider" on page 223

Related tasks:

- "Updating and Deleting Retrieved Data in Static SQL Programs" on page 92

Example of a Fetch in a Static SQL Program

The following sample selects from a table using a cursor, opens the cursor, and fetches rows from the table. For each row fetched, the program decides, based on simple criteria, whether the row should be deleted or updated.

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

- C/C++ (**tut_mod.sqc/tut_mod.sqC**)

  The following example is from the sample **tut_mod**. This example selects from a table using a cursor, opens the cursor, fetches, updates, or delete rows from the table, then closes the cursor.

```
EXEC SQL DECLARE c1 CURSOR FOR SELECT * FROM staff WHERE id >= 310;
  EXEC SQL OPEN c1;
  EXEC SQL FETCH c1 INTO :id, :name, :dept, :job:jobInd, :years:yearsInd, :salary,
     :comm:commInd;
```

  The sample **tbmod** is a longer version of the **tut_mod** sample, and shows almost all possible cases of table data modification.

- Java™ (**TutMod.sqlj**)

  The following example is from the sample **TutMod**. This example selects from a table using a cursor, opens the cursor, fetches, updates, or delete rows from the table, then closes the cursor.

```
#sql cur = {SELECT * FROM staff WHERE id >= 310};
#sql {FETCH :cur INTO :id, :name, :dept, :job, :years, :salary, :comm};
```

  The sample **TbMod** is a longer version of **TutMod** sample, and shows almost all possible cases of table data modification.

- COBOL (**openftch.sqb**)

  The following example is from the sample **openftch**. This example selects from a table using a cursor, opens the cursor, and fetches rows from the table.

```
EXEC SQL DECLARE c1 CURSOR FOR
  SELECT name, dept FROM staff
  WHERE job='Mgr'
  FOR UPDATE OF job END-EXEC.

EXEC SQL OPEN c1 END-EXEC


* call the FETCH and UPDATE/DELETE loop.
  perform Fetch-Loop thru End-Fetch-Loop
  until SQLCODE not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
```

Related concepts:

- "Error Message Retrieval in an Application" on page 102

Related samples:

- "openftch.sqb -- How to modify table data using cursor statically (IBM COBOL)"
- "tbmod.sqc -- How to modify table data (C)"
- "tut_mod.out -- HOW TO MODIFY TABLE DATA (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "tbmod.sqC -- How to modify table data (C++)"
- "tut_mod.out -- HOW TO MODIFY TABLE DATA (C++)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "TbMod.sqlj -- How to modify table data (SQLj)"
- "TutMod.out -- HOW TO MODIFY TABLE DATA. Connect to 'sample' database using JDBC type 2 driver (SQLJ)"
- "TutMod.sqlj -- Modify data in a table (SQLj)"

Scrolling Through and Manipulating Retrieved Data

The sections that follow describe how to scroll through retrieved data. The sample programs that show how to manipulate data are also briefly described.

Scrolling Through Previously Retrieved Data

When an application retrieves data from the database, the FETCH statement allows it to scroll forward through the data, however, the database manager has no embedded SQL statement that allows it scroll backwards through the data, (equivalent to a backward FETCH). DB2 CLI and Java, however, do support a backward FETCH through read-only scrollable cursors.

Procedure:

For embedded SQL applications, you can use the following techniques to scroll through data that has been retrieved:

- Keep a copy of the data that has been fetched and scroll through it by some programming technique.
- Use SQL to retrieve the data again, typically by a second SELECT statement.

Related tasks:

- "Keeping a Copy of the Data" on page 95
- "Retrieving Data a Second Time" on page 95

Related reference:

- "SQLFetchScroll function (CLI) - Fetch rowset and return data for all bound columns" in the *CLI Guide and Reference, Volume 2*
- "Cursor positioning rules for SQLFetchScroll() (CLI)" in the *CLI Guide and Reference, Volume 2*

Keeping a Copy of the Data

In some situations, it may be useful to maintain a copy of data that is fetched by the application.

Procedure:

To keep a copy of the data, your application can do the following:

- Save the fetched data in virtual storage.
- Write the data to a temporary file (if the data does not fit in virtual storage). One effect of this approach is that a user, scrolling backward, always sees exactly the same data that was fetched, even if the data in the database was changed in the interim by a transaction.
- Using an isolation level of repeatable read, the data you retrieve from a transaction can be retrieved again by closing and opening a cursor. Other applications are prevented from updating the data in your result set. Isolation levels and locking can affect how users update data.

Related concepts:

- "Row Order Differences Between the First and Second Result Table" on page 96

Related tasks:

- "Retrieving Data a Second Time" on page 95

Retrieving Data a Second Time

The technique that you use to retrieve data a second time depends on the order in which you want to see the data again.

Procedure:

You can retrieve data a second time by using any of the following methods:

- Retrieve data from the beginning

  To retrieve the data again from the beginning of the result table, close the active cursor and reopen it. This action positions the cursor at the beginning of the result table. But, unless the application holds locks on the table, others may have changed it, so what had been the first row of the result table may no longer be.
- Retrieve data from the middle

  To retrieve data a second time from somewhere in the middle of the result table, execute a second SELECT statement and declare a second cursor on the statement. For example, suppose the first SELECT statement was:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

Now, suppose that you want to return to the rows that start with `DEPTNO = 'M95'` and fetch sequentially from that point. Code the following:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'M95'
  ORDER BY DEPTNO
```

This statement positions the cursor where you want it.

- Retrieve data in reverse order

  Ascending ordering of rows is the default. If there is only one row for each value of `DEPTNO`, then the following statement specifies a unique ascending ordering of rows:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    ORDER BY DEPTNO
  ```

  To retrieve the same rows in reverse order, specify that the order is descending, as in the following statement:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    ORDER BY DEPTNO DESC
  ```

  A cursor on the second statement retrieves rows in exactly the opposite order from a cursor on the first statement. Order of retrieval is guaranteed only if the first statement specifies a unique ordering sequence.

  For retrieving rows in reverse order, it can be useful to have two indexes on the `DEPTNO` column, one in ascending order, and the other in descending order.

Related concepts:

- "Row Order Differences Between the First and Second Result Table" on page 96

Row Order Differences Between the First and Second Result Table

The rows of the second result table may not be displayed in the same order as in the first. The database manager does not consider the order of rows as significant unless the SELECT statement uses ORDER BY. Thus, if there are several rows with the same `DEPTNO` value, the second SELECT statement may retrieve them in a different order from the first. The only guarantee is that they will all be in order by department number, as demanded by the clause ORDER BY DEPTNO.

The difference in ordering could occur even if you were to execute the same SQL statement, with the same host variables, a second time. For example, the statistics in the catalog could be updated between executions, or indexes could be created or dropped. You could then execute the SELECT statement again.

The ordering is more likely to change if the second SELECT has a predicate that the first did not have; the database manager could choose to use an index on the new predicate. For example, it could choose an index on `LOCATION` for the first statement in our example, and an index on `DEPTNO` for the second. Because rows are fetched in order by the index key, the second order need not be the same as the first.

Again, executing two similar SELECT statements can produce a different ordering of rows, even if no statistics change and no indexes are created or dropped. In the example, if there are many different values of LOCATION, the database manager could choose an index on LOCATION for both statements. Yet changing the value of DEPTNO in the second statement to the following, could cause the database manager to choose an index on DEPTNO:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

Because of the subtle relationships between the form of an SQL statement and the values in this statement, never assume that two different SQL statements will return rows in the same order unless the order is uniquely determined by an ORDER BY clause.

Related tasks:

- "Retrieving Data a Second Time" on page 95

Positioning a Cursor at the End of a Table

If you need to position the cursor at the end of a table, you can use an SQL statement to position it.

Procedure:

Use either of the following examples as a method for positioning a cursor:

- The database manager does not guarantee an order to data stored in a table; therefore, the end of a table is not defined. However, order is defined on the result of an SQL statement:

  ```
  SELECT * FROM DEPARTMENT
    ORDER BY DEPTNO DESC
  ```

- The following statement positions the cursor at the row with the highest DEPTNO value:

  ```
  SELECT * FROM DEPARTMENT
    WHERE DEPTNO =
    (SELECT MAX(DEPTNO) FROM DEPARTMENT)
  ```

Note, however, that if several rows have the same value, the cursor is positioned on the first of them.

Updating Previously Retrieved Data

To scroll backward and update data that was retrieved previously, you can use a combination of the techniques that are used to scroll through previously retrieved data and to update retrieved data.

Procedure:

To update previously retrieved data, you can do one of two things:

- If you have a second cursor on the data to be updated and the SELECT statement uses none of the restricted elements, you can use a cursor-controlled UPDATE statement. Name the second cursor in the WHERE CURRENT OF clause.

- In other cases, use UPDATE with a WHERE clause that names all the values in the row or specifies the primary key of the table. You can execute one statement many times with different values of the variables.

Related tasks:

- "Updating and Deleting Retrieved Data in Static SQL Programs" on page 92
- "Scrolling Through Previously Retrieved Data" on page 94

Example of an Insert, Update, and Delete in a Static SQL Program

The following examples show how to insert, update, and delete data using static SQL.

- C/C++ (**tut_mod.sqc/tut_mod.sqC**)

  The following three examples are from the **tut_mod** sample. See this sample for a complete program that shows how to modify table data in C or C++.

  The following example shows how to insert table data:

```
EXEC SQL INSERT INTO staff(id, name, dept, job, salary)
  VALUES(380, 'Pearce', 38, 'Clerk', 13217.50),
        (390, 'Hachey', 38, 'Mgr', 21270.00),
        (400, 'Wagland', 38, 'Clerk', 14575.00);
```

  The following example shows how to update table data:

```
EXEC SQL UPDATE staff
  SET salary = salary + 10000
  WHERE id >= 310 AND dept = 84;
```

  The following example shows how to delete from a table:

```
EXEC SQL DELETE
  FROM staff
  WHERE id >= 310 AND salary > 20000;
```

- Java™ (**TutMod.sqlj**)

  The following three examples are from in the **TutMod** sample. See this sample for a complete program that shows how to modify table data in SQLJ.

  The following example shows how to insert table data:

```
#sql {INSERT INTO staff(id, name, dept, job, salary)
        VALUES(380, 'Pearce', 38, 'Clerk', 13217.50),
              (390, 'Hachey', 38, 'Mgr', 21270.00),
              (400, 'Wagland', 38, 'Clerk', 14575.00)};
```

  The following example shows how to update table data:

```
#sql {UPDATE staff
        SET salary = salary + 1000
        WHERE id >= 310 AND dept = 84};
```

  The following example shows how to delete from a table:

```
#sql {DELETE FROM staff
        WHERE id >= 310 AND salary > 20000};
```

- COBOL (**updat.sqb**)

  The following three examples are from the **updat** sample. See this sample for a complete program that shows how to modify table data in COBOL.

  The following example shows how to insert table data:

```
EXEC SQL INSERT INTO staff
    VALUES (999, 'Testing', 99, :job-update, 0, 0, 0)
    END-EXEC.
```

  The following example shows how to update table data:

```
EXEC SQL UPDATE staff
    SET job=:job-update
    WHERE job='Mgr'
    END-EXEC.
```

The following example shows how to delete from a table:

```
EXEC SQL DELETE
    FROM staff
    WHERE job=:job-update
    END-EXEC.
```

Related concepts:

- "Error Message Retrieval in an Application" on page 102

Related samples:

- "tbinfo.out -- HOW TO GET INFORMATION AT THE TABLE LEVEL (C++)"
- "tbmod.out -- HOW TO MODIFY TABLE DATA (C++)"
- "tbmod.sqC -- How to modify table data (C++)"
- "tut_mod.out -- HOW TO MODIFY TABLE DATA (C++)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "tbmod.out -- HOW TO MODIFY TABLE DATA (C)"
- "tbmod.sqc -- How to modify table data (C)"
- "tut_mod.out -- HOW TO MODIFY TABLE DATA (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "TbMod.out -- HOW TO MODIFY TABLE DATA. Connect to 'sample' database using JDBC type 2 driver (SQLJ)"
- "TbMod.sqlj -- How to modify table data (SQLj)"
- "TutMod.out -- HOW TO MODIFY TABLE DATA. Connect to 'sample' database using JDBC type 2 driver (SQLJ)"
- "TutMod.sqlj -- Modify data in a table (SQLj)"

Diagnostic Information

The sections that follow describe the diagnostic information that is available for a static SQL program, such as return codes and how an application should retrieve error messages.

Return Codes

Most database manager APIs pass back a zero return code when successful. In general, a non-zero return code indicates that the secondary error handling mechanism, the SQLCA structure, may be corrupt. In this case, the called API is not executed. A possible cause for a corrupt SQLCA structure is passing an invalid address for the structure.

Related reference:

- "SQLCA" in the *Administrative API Reference*

Error Information in the SQLCODE, SQLSTATE, and SQLWARN Fields

Error information is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure, which is updated after every executable SQL statement and most database manager API calls.

A source file containing executable SQL statements can provide at least one SQLCA structure with the name `sqlca`. The SQLCA structure is defined in the SQLCA include file. Source files without embedded SQL statements, but calling database manager APIs, can also provide one or more SQLCA structures, but their names are arbitrary.

If your application is compliant with the FIPS 127-2 standard, you can declare the SQLSTATE and SQLCODE as host variables for C, C++, COBOL, and FORTRAN applications, instead of using the SQLCA structure.

An SQLCODE value of 0 means successful execution (with possible SQLWARN warning conditions). A positive value means that the statement was successfully executed but with a warning, as with truncation of a host variable. A negative value means that an error condition occurred.

An additional field, SQLSTATE, contains a standardized error code consistent across other IBM® database products and across SQL92–conformant database managers. Practically speaking, you should use SQLSTATE values when you are concerned about portability since SQLSTATE values are common across many database managers.

The SQLWARN field contains an array of warning indicators, even if SQLCODE is zero. The first element of the SQLWARN array, SQLWARN0, contains a blank if all other elements are blank. SQLWARN0 contains a `W` if at least one other element contains a warning character.

Note: If you want to develop applications that access various IBM RDBMS servers you should:

- Where possible, have your applications check the SQLSTATE rather than the SQLCODE.
- If your applications will use DB2 Connect, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.

Related concepts:

- "Return Codes" on page 99
- "SQLSTATE and SQLCODE Variables in C and C++" on page 168
- "SQLSTATE and SQLCODE Variables in COBOL" on page 193
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 208
- "SQLSTATE and SQLCODE Variables in Perl" on page 491

Related reference:

- "SQLCA" in the *Administrative API Reference*

Token Truncation in the SQLCA Structure

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

Related reference:

- "SQLCA" in the *Administrative API Reference*

Exception, Signal, and Interrupt Handler Considerations

An exception, signal, or interrupt handler is a routine that gets control when an exception, signal, or interrupt occurs. The type of handler applicable is determined by your operating environment, as shown in the following:

Windows® operating systems
Pressing `Ctrl-C` or `Ctrl-Break` generates an interrupt.

UNIX®-based systems
Usually, pressing `Ctrl-C` generates the SIGINT interrupt signal. Note that keyboards can easily be redefined so SIGINT may be generated by a different key sequence on your machine.

Do not put SQL statements (other than COMMIT or ROLLBACK) in exception, signal, and interrupt handlers. With these kinds of error conditions, you normally want to do a ROLLBACK to avoid the risk of inconsistent data.

Note that you should exercise caution when coding a COMMIT and ROLLBACK in exception/signal/interrupt handlers. If you call either of these statements by themselves, the COMMIT or ROLLBACK is not executed until the current SQL statement is complete, if one is running. This is not the behavior desired from a `Ctrl-C` handler.

The solution is to call the INTERRUPT API (`sqleintr/sqlgintr`) before issuing a ROLLBACK. This API interrupts the current SQL query (if the application is executing one) and lets the ROLLBACK begin immediately. If you are going to perform a COMMIT rather than a ROLLBACK, you do not want to interrupt the current command.

When using APPC to access a remote database server (DB2 for AIX or host database system using DB2 Connect), the application may receive a SIGUSR1 signal. This signal is generated by SNA Services/6000 when an unrecoverable error occurs and the SNA connection is stopped. You may want to install a signal handler in your application to handle SIGUSR1.

Refer to your platform documentation for specific details on the various handler considerations.

Related concepts:

- "Processing of Interrupt Requests" on page 694

Exit List Routine Considerations

Do not use SQL or DB2 API calls in exit list routines. Note that you cannot disconnect from a database in an exit routine.

Error Message Retrieval in an Application

Depending on the language in which your application is written, you use a different method to retrieve error information:

- C, C++, and COBOL applications can use the GET ERROR MESSAGE API to obtain the corresponding information related to the SQLCA passed in.
- JDBC and SQLJ applications throw an SQLException when an error occurs during SQL processing. Your applications can catch and display an SQLException with the following code:

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

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

- REXX applications use the CHECKERR procedure.

Related concepts:

- "SQLSTATE and SQLCODE Variables in C and C++" on page 168
- "SQLSTATE and SQLCODE Variables in COBOL" on page 193
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 208
- "SQLSTATE and SQLCODE Variables in Perl" on page 491

Related reference:

- "sqlaintp - Get Error Message" in the *Administrative API Reference*

Chapter 5. Writing Dynamic SQL Programs

Characteristics and Reasons for Using Dynamic SQL

The sections that follow describe the reasons for using dynamic SQL as compared to static SQL.

Reasons for Using Dynamic SQL

You may want to use dynamic SQL when:

- You need all or part of the SQL statement to be generated during application execution.
- The objects referenced by the SQL statement do not exist at precompile time.
- You want the statement to always use the most optimal access path, based on current database statistics.
- You want to modify the compilation environment of the statement, that is, experiment with the special registers.

Related concepts:

- "Dynamic SQL Support Statements" on page 103
- "Dynamic SQL Versus Static SQL" on page 104

Dynamic SQL Support Statements

The dynamic SQL support statements accept a character-string host variable and a statement name as arguments. The host variable contains the SQL statement to be processed dynamically in text form. The statement text is not processed when an

application is precompiled. In fact, the statement text does not have to exist at the time the application is precompiled. Instead, the SQL statement is treated as a host variable for precompilation purposes and the variable is referenced during application execution. These SQL statements are referred to as *dynamic* SQL.

Dynamic SQL support statements are required to transform the host variable containing SQL text into an executable form and operate on it by referencing the statement name. These statements are:

EXECUTE IMMEDIATE
Prepares and executes a statement that does not use any host variables. All EXECUTE IMMEDIATE statements in an application are cached in the same place at run time, so only the last statement is known. Use this statement as an alternative to the PREPARE and EXECUTE statements.

PREPARE
Turns the character string form of the SQL statement into an executable form of the statement, assigns a statement name, and optionally places information about the statement in an SQLDA structure.

EXECUTE
Executes a previously prepared SQL statement. The statement can be executed repeatedly within a connection.

DESCRIBE
Places information about a prepared statement into an SQLDA.

An application can execute most supported SQL statements dynamically.

Note: The content of dynamic SQL statements follows the same syntax as static SQL statements, with the following exceptions:

- Comments are not allowed.
- The statement cannot begin with EXEC SQL.
- The statement cannot end with the statement terminator. An exception to this is the CREATE TRIGGER statement which can contain a semicolon (;).

Related reference:

- Appendix A, “Supported SQL Statements,” on page 685

Dynamic SQL Versus Static SQL

The question of whether to use static or dynamic SQL for performance is usually of great interest to programmers. The answer depends on your situation.

Use the following table when deciding whether to use static or dynamic SQL. Considerations such as security dictate static SQL, while environmental considerations (for example, using DB2 CLI or the CLP) dictate dynamic SQL. When making your decision, consider the following recommendations on whether to choose static or dynamic SQL in a particular situation. In the following table, 'Either' means that there is no advantage to either static or dynamic SQL.

Note: These are general recommendations only. Your specific application, its intended usage, and working environment dictate the actual choice. When in doubt, prototyping your statements as static SQL, then as dynamic SQL, then comparing the differences is the best approach.

Table 8. Comparing Static and Dynamic SQL

Consideration	Likely Best Choice
Time to run the SQL statement:	
• Less than 2 seconds	• Static
• 2 to 10 seconds	• Either
• More than 10 seconds	• Dynamic
Data Uniformity	
• Uniform data distribution	• Static
• Slight non-uniformity	• Either
• Highly non-uniform distribution	• Dynamic
Range (<,>,BETWEEN,LIKE) Predicates	
• Very Infrequent	• Static
• Occasional	• Either
• Frequent	• Dynamic
Repetitious Execution	
• Runs many times (10 or more times)	• Either
• Runs a few times (less than 10 times)	• Either
• Runs once	• Static
Nature of Query	
• Random	• Dynamic
• Permanent	• Either
Run Time Environment (DML/DDL)	
• Transaction Processing (DML Only)	• Either
• Mixed (DML and DDL - DDL affects packages)	• Dynamic
• Mixed (DML and DDL - DDL does not affect packages)	• Either
Frequency of `RUNSTATS`	
• Very infrequently	• Static
• Regularly	• Either
• Frequently	• Dynamic

In general, an application using dynamic SQL has a higher start-up (or initial) cost per SQL statement due to the need to compile the SQL statements before using them. Once compiled, the execution time for dynamic SQL compared to static SQL should be equivalent and, in some cases, faster due to better access plans being chosen by the optimizer. Each time a dynamic statement is executed, the initial compilation cost becomes less of a factor. If multiple users are running the same dynamic application with the same statements, only the first application to issue the statement realizes the cost of statement compilation.

In a mixed DML and DDL environment, the compilation cost for a dynamic SQL statement may vary as the statement may be implicitly recompiled by the system while the application is running. In a mixed environment, the choice between static and dynamic SQL must also factor in the frequency in which packages are invalidated. If the DDL does invalidate packages, dynamic SQL may be more efficient as only those queries executed are recompiled when they are next used. Others are not recompiled. For static SQL, the entire package is rebound once it has been invalidated.

Now suppose your particular application contains a mixture of the above characteristics, and some of these characteristics suggest that you use static while others suggest dynamic. In this case, there is no obvious decision, and you should

probably use the method you have the most experience with, and with which you feel most comfortable. Note that the considerations in the above table are listed roughly in order of importance.

Note: Static and dynamic SQL each come in two types that make a difference to the DB2 optimizer. These types are:

1. Static SQL containing no host variables

   This is an unlikely situation which you may see only for:

   - *Initialization* code
   - Novice training examples

   This is actually the best combination from a performance perspective in that there is no run-time performance overhead, and the DB2 optimizer's capabilities can be fully realized.

2. Static SQL containing host variables

   This is the traditional *legacy* style of DB2® applications. It avoids the run time overhead of a PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be utilized because the optimizer does not know the entire SQL statement. A particular problem exists with highly non-uniform data distributions.

3. Dynamic SQL containing no parameter markers

   This is the typical style for random query interfaces (such as the CLP), and is the optimizer's preferred *flavor* of SQL. For complex queries, the overhead of the PREPARE statement is usually offset by the improved execution time.

4. Dynamic SQL containing parameter markers

   This is the most common type of SQL for CLI applications. The key benefit is that the presence of parameter markers allows the cost of the PREPARE to be amortized over the repeated executions of the statement, typically a select or insert. This amortization is true for all repetitive dynamic SQL applications. Unfortunately, just like static SQL with host variables, parts of the DB2 optimizer will not work because complete information is unavailable. The recommendation is to use *static SQL with host variables* or *dynamic SQL without parameter markers* as the most efficient options.

Related concepts:

- "Example of Parameter Markers in a Dynamic SQL Program" on page 125

Related tasks:

- "Providing Variable Input to Dynamic SQL Using Parameter Markers" on page 124

Cursors in Dynamic SQL Programs

The sections that follow describe how to declare and use cursors in dynamic SQL, and briefly describe the sample programs that use cursors.

Declaring and Using Cursors in Dynamic SQL Programs

Processing a cursor dynamically is nearly identical to processing it using static SQL. When a cursor is declared, it is associated with a query.

In static SQL, the query is a SELECT statement in text form, while in dynamic SQL, the query is associated with a statement name assigned in a PREPARE statement. Any referenced host variables are represented by parameter markers.

The main difference between a static and a dynamic cursor is that a static cursor is prepared at precompile time, and a dynamic cursor is prepared at run time. Additionally, host variables referenced in the query are represented by parameter markers, which are replaced by run-time host variables when the cursor is opened.

Procedure:

Use the examples shown in the following table when coding cursors for a dynamic SQL program:

Table 9. Declare Statement Associated with a Dynamic SELECT

Language	Example Source Code

C/C++	`strcpy( prep_string, "SELECT tabname FROM syscat.tables"`
	`"WHERE tabschema = ?" );`
	`EXEC SQL PREPARE s1 FROM :prep_string;`
	`EXEC SQL DECLARE c1 CURSOR FOR s1;`
	`EXEC SQL OPEN c1 USING :host_var;`

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

COBOL	`MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`
	`TO PREP-STRING.`
	`EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC.`
	`EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.`
	`EXEC SQL OPEN C1 USING :host-var END-EXEC.`

FORTRAN	`prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?'`
	`EXEC SQL PREPARE s1 FROM :prep_string`
	`EXEC SQL DECLARE c1 CURSOR FOR s1`
	`EXEC SQL OPEN c1 USING :host_var`

Related concepts:

- "Example of a Cursor in a Dynamic SQL Program" on page 107
- "Cursors in REXX" on page 502

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 87

Example of a Cursor in a Dynamic SQL Program

A dynamic SQL statement can be prepared for execution with the PREPARE statement and executed with the EXECUTE statement or the DECLARE CURSOR statement.

PREPARE with EXECUTE

The following example shows how a dynamic SQL statement can be prepared for execution with the PREPARE statement and executed with the EXECUTE statement:

- C/C++ (**dbuse.sqc/dbuse.sqC**):

The following example is from the sample **dbuse**:

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt[50];
EXEC SQL END DECLARE SECTION;

strcpy(hostVarStmt, "DELETE FROM org WHERE deptnumb = 15");
EXEC SQL PREPARE Stmt FROM :hostVarStmt;
EXEC SQL EXECUTE Stmt;
```

PREPARE with DECLARE CURSOR

The following examples show how a dynamic SQL statement can be prepared for execution with the PREPARE statement, and executed with the DECLARE CURSOR statement:

- C

```
EXEC SQL BEGIN DECLARE SECTION;
  char  st[80];
  char  parm_var[19};
EXEC SQL END DECLARE SECTION;

strcpy( st, "SELECT tabname FROM syscat.tables" );
strcat( st, " WHERE tabname <> ? ORDER BY 1" );
EXEC SQL PREPARE s1 FROM :st;
EXEC SQL DECLARE c1 CURSOR FOR s1;
strcpy( parm_var, "STAFF" );
EXEC SQL OPEN c1 USING :parm_var;
```

- Java™

```
PreparedStatement pstmt1 = con.prepareStatement(
  "SELECT tabname FROM syscat.tables " +
  "WHERE tabname <> ? ORDER BY 1");

// set cursor name for the positioned update statement
pstmt1.setCursorName("c1");
pstmt1.setString(1, "STAFF");
ResultSet rs = pstmt1.executeQuery();
```

- COBOL **(dynamic.sqb**)

  The following example is from the **dynamic.sqb** sample:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     01 st              pic x(80).
     01 parm-var        pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

move "SELECT TABNAME FROM SYSCAT.TABLES ORDER BY 1 WHERE TABNAME <> ?" to st.
EXEC SQL PREPARE s1 FROM :st END-EXEC.

EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

move "STAFF" to parm-var.
EXEC SQL OPEN c1 USING :parm-var END-EXEC.
```

EXECUTE IMMEDIATE

You can can also prepare and execute a dynamic SQL satement with the EXECUTE IMMEDIATE statement (except for SELECT statements that return more than one row).

- C/C++ (**dbuse.sqc/dbuse.sqC**)

  The following example is from the function `DynamicStmtEXECUTE_IMMEDIATE()` in the sample **dbuse**:

```
EXEC SQL BEGIN DECLARE SECTION;
  char stmt1[50];
EXEC SQL END DECLARE SECTION;

strcpy(stmt1, "CREATE TABLE table1(col1 INTEGER)");
EXEC SQL EXECUTE IMMEDIATE :stmt1;
```

Related concepts:

- "Error Message Retrieval in an Application" on page 102

Related samples:

- "dbuse.out -- HOW TO USE A DATABASE (C)"
- "dbuse.sqc -- How to use a database (C)"
- "dbuse.out -- HOW TO USE A DATABASE (C++)"
- "dbuse.sqC -- How to use a database (C++)"

Effects of REOPT on dynamic SQL

When you specify the option REOPT ALWAYS, DB2® postpones preparing any statement containing host variables, parameter markers, or special registers until it encounters an OPEN or EXECUTE statement; that is, when the values for these variables become known. At this time, the access plan is generated using these values. Subsequent OPEN or EXECUTE requests for the same statement will recompile the statement, reoptimize the query plan using the current set of values for the variables, and execute the newly generated query plan.

The option REOPT ONCE has a similar effect, with the exception that the plan is only optimized once using the values of the host variables, parameter markers and special registers. This plan is cached and will be used by subsequent requests.

Effect of DYNAMICRULES bind option on dynamic SQL

The PRECOMPILE and BIND option DYNAMICRULES determines what values apply at run-time for the following dynamic SQL attributes:

- The authorization ID that is used during authorization checking.
- The qualifier that is used for qualification of unqualified objects.
- Whether the package can be used to dynamically prepare the following statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE statements.

In addition to the DYNAMICRULES value, the run-time environment of a package controls how dynamic SQL statements behave at run-time. The two possible run-time environments are:

- The package runs as part of a stand-alone program
- The package runs within a routine context

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

Run behavior DB2® uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization checking of dynamic SQL statements and for the

initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

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

Define behavior

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

Invoke behavior

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

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

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

Table 10. How DYNAMICRULES and the Run-Time Environment Determine Dynamic SQL Statement Behavior

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

Table 10. How DYNAMICRULES and the Run-Time Environment Determine Dynamic SQL Statement Behavior (continued)

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

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

Table 11. Definitions of Dynamic SQL Statement Behaviors

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

Related concepts:

- "Authorization Considerations for Dynamic SQL" on page 47
- "Authorizations and binding of routines that contain SQL" in the *Application Development Guide: Programming Server Applications*

The SQLDA in Dynamic SQL Programs

The sections that follow describe the different considerations that apply when you declare the SQLDA for a dynamic SQL program.

Host Variables and the SQLDA in Dynamic SQL Programs

With static SQL, host variables used in embedded SQL statements are known at application compile time. With dynamic SQL, the embedded SQL statements and consequently the host variables are not known until application run time. Thus, for dynamic SQL applications, you need to deal with the list of host variables that are used in your application. You can use the DESCRIBE statement to obtain host

variable information for any SELECT statement that has been prepared (using PREPARE), and store that information into the SQL descriptor area (SQLDA).

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

When the DESCRIBE statement gets executed in your application, the database manager defines your host variables in an SQLDA. Once the host variables are defined in the SQLDA, you can use the FETCH statement to assign values to the host variables, using a cursor.

Related concepts:

- "Example of a Cursor in a Dynamic SQL Program" on page 107

Related reference:

- "DESCRIBE statement" in the *SQL Reference, Volume 2*
- "FETCH statement" in the *SQL Reference, Volume 2*
- "PREPARE statement" in the *SQL Reference, Volume 2*
- "SQLDA" in the *Administrative API Reference*

Declaring the SQLDA Structure in a Dynamic SQL Program

An SQLDA contains a variable number of occurrences of SQLVAR entries, each of which contains a set of fields that describe one column in a row of data, as shown in the following figure. There are two types of SQLVAR entries: base SQLVAR entries, and secondary SQLVAR entries.

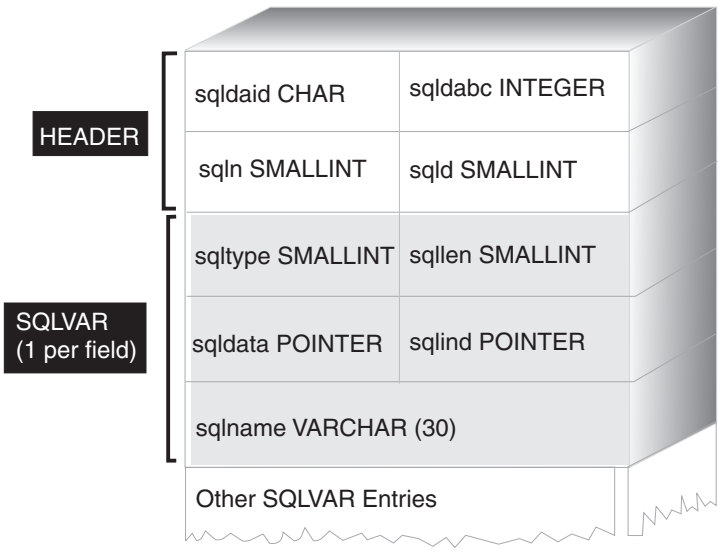


Figure 3. The SQL Descriptor Area (SQLDA)

Procedure:

Because the number of SQLVAR entries required depends on the number of columns in the result table, an application must be able to allocate an appropriate number of SQLVAR elements when needed. Use one of the following methods:

- Provide the largest SQLDA (that is, the one with the greatest number of SQLVAR entries) that is needed. The maximum number of columns that can be returned in a result table is 255. If any of the columns being returned is either a LOB type or a distinct type, the value in SQLN is doubled, and the number of SQLVAR entries needed to hold the information is doubled to 510. However, as most SELECT statements do not even retrieve 255 columns, most of the allocated space is unused.
- Provide a smaller SQLDA with fewer SQLVAR entries. In this case, if there are more columns in the result than SQLVAR entries allowed for in the SQLDA, no descriptions are returned. Instead, the database manager returns the number of select list items detected in the SELECT statement. The application allocates an SQLDA with the required number of SQLVAR entries, then uses the DESCRIBE statement to acquire the column descriptions.

For both methods, the question arises as to how many initial SQLVAR entries you should allocate. Each SQLVAR element uses up 44 bytes of storage (not counting storage allocated for the SQLDATA and SQLIND fields). If memory is plentiful, the first method of providing an SQLDA of maximum size is easier to implement.

The second method of allocating a smaller SQLDA is only applicable to programming languages such as C and C++ that support the dynamic allocation of memory. For languages such as COBOL and FORTRAN that do not support the dynamic allocation of memory, you have to use the first method.

Related tasks:

- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 113
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 115
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 117

Related reference:

- "SQLDA" in the *Administrative API Reference*

Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure

Use the information provided here as an example of how to allocate the minimum SQLDA structure for a statement.

Restrictions:

You can only allocate a smaller SQLDA structure with programming languages, such as C and C++, that support the dynamic allocation of memory.

Procedure:

Suppose an application declares an SQLDA structure named `minsqlda` that contains no SQLVAR entries. The SQLN field of the SQLDA describes the number of SQLVAR entries that are allocated. In this case, SQLN must be set to 0. Next, to prepare a statement from the character string `dstring` and to enter its description

into `minsqlda`, issue the following SQL statement (assuming C syntax, and assuming that `minsqlda` is declared as a pointer to an SQLDA structure):

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

Suppose that the statement contained in `dstring` is a SELECT statement that returns 20 columns in each row. After the PREPARE statement (or a DESCRIBE statement), the SQLD field of the SQLDA contains the number of columns of the result table for the prepared SELECT statement.

The SQLVAR entries in the SQLDA are set in the following cases:

- SQLN >= SQLD and no column is either a LOB or a distinct type.

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.
- SQLN >= 2*SQLD and at least one column is a LOB or a distinct type.

  2* SQLD SQLVAR entries are set and SQLDOUBLED is set to 2.
- SQLD <= SQLN < 2*SQLD and at least one column is a distinct type, but there are no LOB columns.

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVAR entries in the SQLDA are *not* set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no column is either a LOB or distinct type.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVAR entries for a successful DESCRIBE.
- SQLN < SQLD and at least one column is a distinct type, but there are no LOB columns.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

  Allocate 2*SQLD SQLVAR entries for a successful DESCRIBE, including the names of the distinct types.
- SQLN < 2*SQLD and at least one column is a LOB.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVAR entries for a successful DESCRIBE.

The SQLWARN option of the BIND command is used to control whether the DESCRIBE (or PREPARE...INTO) will return the following warnings:

- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005).

It is recommended that your application code always consider that these SQLCODE values could be returned. The warning SQLCODE +238 (SQLSTATE 01005) is always returned when there are LOB columns in the select list and there are insufficient SQLVAR entries in the SQLDA. This is the only way the application can know that the number of SQLVAR entries must be doubled because of a LOB column in the result set.

Related tasks:

- "Declaring the SQLDA Structure in a Dynamic SQL Program" on page 112
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 115
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 117

Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program

After you determine the number of columns in the result table, allocate storage for a second, full-size SQLDA.

Procedure:

Assume that the result table contains 20 columns (none of which are LOB columns). In this situation, you must allocate a second SQLDA structure, `fulsqlda` with at least 20 SQLVAR elements (or 40 elements if the result table contains any LOBs or distinct types). For the rest of this example, assume that no LOBs or distinct types are in the result table.

When you calculate the storage requirements for SQLDA structures, include the following:

- A fixed-length header, 16 bytes in length, containing fields such as SQLN and SQLD
- A variable-length array of SQLVAR entries, of which each element is 44 bytes in length on 32-bit platforms, and 56 bytes in length on 64-bit platforms.

The number of SQLVAR entries needed for `fulsqlda` is specified in the SQLD field of `minsqlda`. Assume this value is 20. Therefore, the storage allocation required for `fulsqlda` is:

```
16 + (20 * sizeof(struct sqlvar))
```

Note: On 64-bit platforms, `sizeof(struct sqlvar)` and `sizeof(struct sqlvar2)` returns 56. On 32-bit platforms, `sizeof(struct sqlvar)` and `sizeof(struct sqlvar2)` returns 44.

This value represents the size of the header plus 20 times the size of each SQLVAR entry, giving a total of 896 bytes.

You can use the SQLDASIZE macro to avoid doing your own calculations and to avoid any version-specific dependencies.

Related tasks:

- "Declaring the SQLDA Structure in a Dynamic SQL Program" on page 112
- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 113
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 117

Describing a SELECT Statement in a Dynamic SQL Program

After you allocate sufficient space for the second SQLDA (in this example, called `fulsqlda`), you must code the application to describe the SELECT statement.

Procedure:

Code your application to perform the following steps:

1. Store the value `20` in the SQLN field of `fulsqlda` (the assumption in this example is that the result table contains 20 columns, and none of these columns are LOB columns).
2. Obtain information about the SELECT statement using the second SQLDA structure, `fulsqlda`. Two methods are available:
   - Use another PREPARE statement, specifying `fulsqlda` instead of `minsqlda`.
   - Use the DESCRIBE statement specifying `fulsqlda`.

Using the DESCRIBE statement is preferred because the costs of preparing the statement a second time are avoided. The DESCRIBE statement simply reuses information previously obtained during the prepare operation to fill in the new SQLDA structure. The following statement can be issued:

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

After this statement is executed, each SQLVAR element contains a description of one column of the result table.

Related tasks:

- "Acquiring Storage to Hold a Row" on page 116

Acquiring Storage to Hold a Row

Before the application can fetch a row of the result table using an SQLDA structure, the application must first allocate storage for the row.

Procedure:

Code your application to do the following:

1. Analyze each SQLVAR description to determine how much space is required for the value of that column.

   Note that for LOB values, when the SELECT is described, the data type given in the SQLVAR is SQL_TYP_*x*LOB. This data type corresponds to a plain LOB host variable, that is, the whole LOB will be stored in memory at one time. This will work for small LOBs (up to a few MB), but you cannot use this data type for large LOBs (say 1 GB). It will be necessary for your application to change its column definition in the SQLVAR to be either SQL_TYP_*x*LOB_LOCATOR or SQL_TYPE_*x*LOB_FILE. (Note that changing the SQLTYPE field of the SQLVAR also necessitates changing the SQLLEN field.) After changing the column definition in the SQLVAR, your application can then allocate the correct amount of storage for the new type.
2. Allocate storage for the value of that column.
3. Store the address of the allocated storage in the SQLDATA field of the SQLDA structure.

These steps are accomplished by analyzing the description of each column and replacing the content of each SQLDATA field with the address of a storage area large enough to hold any values from that column. The length attribute is determined from the SQLLEN field of each SQLVAR entry for data items that are

not of a LOB type. For items with a type of BLOB, CLOB, or DBCLOB, the length attribute is determined from the SQLLONGLEN field of the secondary SQLVAR entry.

In addition, if the specified column allows nulls, the application must replace the content of the SQLIND field with the address of an indicator variable for the column.

Related concepts:

- "Large object usage" in the *Application Development Guide: Programming Server Applications*

Related tasks:

- "Processing the Cursor in a Dynamic SQL Program" on page 117

Processing the Cursor in a Dynamic SQL Program

After the SQLDA structure is properly allocated, the cursor associated with the SELECT statement can be opened and rows can be fetched.

Procedure:

To process the cursor that is associated with a SELECT statement, first open the cursor, then fetch rows by specifying the USING DESCRIPTOR clause of the FETCH statement. For example, a C application could have the following:

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

For a successful FETCH, you could write the application to obtain the data from the SQLDA and display the column headings. For example:

```
display_col_titles( sqldaPointer ) ;
```

After the data is displayed, you should close the cursor and release any dynamically allocated memory. For example:

```
EXEC SQL CLOSE pcurs ;
EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
```

Allocating an SQLDA Structure for a Dynamic SQL Program

Allocate an SQLDA structure for your application so that you can use it to pass data to and from your application.

Procedure:

To create an SQLDA structure with C, either embed the INCLUDE SQLDA statement in the host language or include the SQLDA include file to get the structure definition. Then, because the size of an SQLDA is not fixed, the application must declare a pointer to an SQLDA structure and allocate storage for it. The actual size of the SQLDA structure depends on the number of distinct data items being passed using the SQLDA.

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

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

On the HP-UX platform, the macro has the following format:

```
#define SQLDASIZE(n) (sizeof(struct sqlda) \
        + (n-1) × sizeof(struct sqlvar))
```

The effect of this macro is to calculate the required storage for an SQLDA with `n` SQLVAR elements.

To create an SQLDA structure with COBOL, you can either embed an INCLUDE SQLDA statement or use the COPY statement. Use the COPY statement when you want to control the maximum number of SQLVAR entries and hence the amount of storage that the SQLDA uses. For example, to change the default number of SQLVAR entries from 1489 to 1, use the following COPY statement:

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

The FORTRAN language does not directly support self-defining data structures or dynamic allocation. No SQLDA include file is provided for FORTRAN, because it is not possible to support the SQLDA as a data structure in FORTRAN. The precompiler will ignore the INCLUDE SQLDA statement in a FORTRAN program.

However, you can create something similar to a static SQLDA structure in a FORTRAN program, and use this structure wherever an SQLDA can be used. The file `sqldact.f` contains constants that help in declaring an SQLDA structure in FORTRAN.

Execute calls to SQLGADDR to assign pointer values to the SQLDA elements that require them.

The following table shows the declaration and use of an SQLDA structure with one SQLVAR element.

Language	Example Source Code

C/C++	`#include <sqlda.h>`
	`struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));`
	
	`/* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */`
	`double sal;`
	`double sal = 0;`
	`short salind;`
	`short salind = 0;`
	
	`/* INITIALIZE ONE ELEMENT OF SQLDA */`
	`memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));`
	`outda->sqln = outda->sqld = 1;`
	`outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;`
	`outda->sqlvar[0].sqllen  = sizeof( double );.`
	`outda->sqlvar[0].sqldata = (unsigned char *)&sal;`
	`outda->sqlvar[0].sqlind  = (short *)&salind;`

Language	Example Source Code

COBOL

```
 WORKING-STORAGE SECTION.
 77 SALARY          PIC S99999V99 COMP-3.
 77 SAL-IND         PIC S9(4)     COMP-5.

    EXEC SQL INCLUDE SQLDA END-EXEC

* Or code a useful way to save unused SQLVAR entries.
* COPY "sqlda.cbl" REPLACING --1489-- BY --1--.

      01 decimal-sqllen pic s9(4) comp-5.
      01 decimal-parts redefines decimal-sqllen.
         05 precision pic x.
         05 scale pic x.

* Initialize one element of output SQLDA
      MOVE 1 TO SQLN
      MOVE 1 TO SQLD
      MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)

* Length = 7 digits precision and 2 digits scale

      MOVE x"07" TO PRECISION.
      MOVE x"02" TO SCALE.
      MOVE DECIMAL-SQLLEN TO O-SQLLEN(1).
      SET SQLDATA(1) TO ADDRESS OF SALARY
      SET SQLIND(1)  TO ADDRESS OF SAL-IND
```

Language	Example Source Code

FORTRAN

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     Declare an Output SQLDA -- 1 Variable
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! Header
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! First Variable
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                                   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                                   )

C     Declare Local Variables for Holding Returned Data.
      real*8       salary
      integer*2    sal_ind

C     Initialize the Output SQLDA (Header)
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     Initialize VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

In languages not supporting dynamic memory allocation, an SQLDA with the desired number of SQLVAR elements must be explicitly declared in the host language. Be sure to declare enough SQLVAR elements as determined by the needs of the application.

Related tasks:

- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 113
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 115
- "Transferring Data in a Dynamic SQL Program Using an SQLDA Structure" on page 121

Transferring Data in a Dynamic SQL Program Using an SQLDA Structure

Greater flexibility is available when transferring data using an SQLDA than is available using lists of host variables. For example, You can use an SQLDA to transfer data that has no native host language equivalent, such as DECIMAL data in the C language.

Procedure:

Use the following table as a cross-reference listing that shows how the numeric values and symbolic names are related.

Table 12. DB2 SQLDA SQL Types. Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name[1]
DATE	384/385	SQL_TYP_DATE / SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME / SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP / SQL_TYP_NSTAMP
n/a[2]	400/401	SQL_TYP_CGSTR / SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB / SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB / SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR / SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG / SQL_TYP_NLONG
n/a[3]	460/461	SQL_TYP_CSTR / SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGRAPH / SQL_TYP_NLONGRAPH
FLOAT	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
REAL[4]	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL[5]	484/485	SQL_TYP_DECIMAL / SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER / SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL / SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE
n/a	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR

Table 12. DB2 SQLDA SQL Types (continued). Numeric Values and Corresponding Symbolic Names

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

Note: These defined types can be found in the `sql.h` include file located in the `include` sub-directory of the `sqllib` directory. (For example, `sqllib/include/sql.h` for the C programming language.)

1. For the COBOL programming language, the SQLTYPE name does not use underscore (_) but uses a hyphen (-) instead.
2. This is a null-terminated graphic string.
3. This is a null-terminated character string.
4. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. Precision is in the first byte. Scale is in the second byte.

Related tasks:

- "Describing a SELECT Statement in a Dynamic SQL Program" on page 115
- "Acquiring Storage to Hold a Row" on page 116
- "Processing the Cursor in a Dynamic SQL Program" on page 117

Processing Interactive SQL Statements in Dynamic SQL Programs

An application using dynamic SQL can be written to process arbitrary SQL statements. For example, if an application accepts SQL statements from a user, the application must be able to execute the statements without any prior knowledge of the statements.

Procedure:

Use the PREPARE and DESCRIBE statements with an SQLDA structure so that the application can determine the type of SQL statement being executed, and act accordingly.

Related concepts:

- "Determination of Statement Type in Dynamic SQL Programs" on page 122

Determination of Statement Type in Dynamic SQL Programs

When an SQL statement is prepared, information concerning the type of statement can be determined by examining the SQLDA structure. This information is placed in the SQLDA structure either at statement preparation time with the INTO clause, or by issuing a DESCRIBE statement against a previously prepared statement.

In either case, the database manager places a value in the SQLD field of the SQLDA structure, indicating the number of columns in the result table generated by the SQL statement. If the SQLD field contains a zero (0), the statement is *not* a SELECT statement. Since the statement is already prepared, it can immediately be executed using the EXECUTE statement.

If the statement contains parameter markers, the USING clause must be specified. The USING clause can specify either a list of host variables or an SQLDA structure.

If the SQLD field is greater than zero, the statement is a SELECT statement and must be processed as described in the following sections.

Related reference:

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

Processing Variable-List SELECT Statements in Dynamic SQL Programs

A *varying-list* SELECT statement is one in which the number and types of columns that are to be returned are not known at precompilation time. In this case, the application does not know in advance the exact host variables that need to be declared to hold a row of the result table.

Procedure:

To process a variable-list SELECT statement, code your application to do the following:

1. Declare an SQLDA.

   An SQLDA structure must be used to process varying-list SELECT statements.
2. PREPARE the statement using the INTO clause.

   The application then determines whether the SQLDA structure declared has enough SQLVAR elements. If it does not, the application allocates another SQLDA structure with the required number of SQLVAR elements, and issues an additional DESCRIBE statement using the new SQLDA.
3. Allocate the SQLVAR elements.

   Allocate storage for the host variables and indicators needed for each SQLVAR. This step involves placing the allocated addresses for the data and indicator variables in each SQLVAR element.
4. Process the SELECT statement.

   A cursor is associated with the prepared statement, opened, and rows are fetched using the properly allocated SQLDA structure.

Related tasks:

- "Declaring the SQLDA Structure in a Dynamic SQL Program" on page 112
- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 113
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 115
- "Describing a SELECT Statement in a Dynamic SQL Program" on page 115
- "Acquiring Storage to Hold a Row" on page 116
- "Processing the Cursor in a Dynamic SQL Program" on page 117

Saving SQL Requests from End Users

If the users of your application can issue SQL requests from the application, you may want to save these requests.

Procedure:

If your application allows users to save arbitrary SQL statements, you can save them in a table with a column having a data type of VARCHAR, LONG VARCHAR, CLOB, VARGRAPHIC, LONG VARGRAPHIC or DBCLOB. Note that

the VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data types are only available in double-byte character set (DBCS) and Extended UNIX Code (EUC) environments.

You must save the source SQL statements, not the prepared versions. This means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your application prepares an SQL statement from a character string and executes this statement dynamically.

Parameter Markers in Dynamic SQL Programs

The sections that follow describe how use parameter markers to provide variable input to a dynamic SQL program, and briefly describe the sample programs that use parameter markers.

Providing Variable Input to Dynamic SQL Using Parameter Markers

A dynamic SQL statement cannot contain host variables, because host variable information (data type and length) is available only during application precompilation. At execution time, the host variable information is not available.

In dynamic SQL, parameter markers are used instead of host variables. Parameter markers are indicated by a question mark (?), and indicate where a host variable is to be substituted inside an SQL statement.

Procedure:

Assume that your application uses dynamic SQL, and that you want to be able to perform a DELETE. A character string containing a parameter marker might look like the following:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

When this statement is executed, a host variable or SQLDA structure is specified by the USING clause of the EXECUTE statement. The contents of the host variable are used when the statement executes.

The parameter marker takes on an assumed data type and length that is dependent on the context of its use inside the SQL statement. If the data type of a parameter marker is not obvious from the context of the statement in which it is used, use a CAST to specify the type. Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like a host variable of the given type. For example, the statement `SELECT ? FROM SYSCAT.TABLES` is not valid because DB2 does not know the type of the result column. However, the statement `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES` is valid because the cast indicates that the parameter marker represents an INTEGER, so DB2 knows the type of the result column.

If the SQL statement contains more than one parameter marker, the USING clause of the EXECUTE statement must either specify a list of host variables (one for each parameter marker), or it must identify an SQLDA that has an SQLVAR entry for each parameter marker. (Note that for LOBs, there are two SQLVAR entries per parameter marker.) The host variable list or SQLVAR entries are matched according to the order of the parameter markers in the statement, and they must have compatible data types.

Note: Using a parameter marker with dynamic SQL is like using host variables with static SQL. In either case, the optimizer does not use distribution statistics, and possibly may not choose the best access plan.

The rules that apply to parameter markers are described with the PREPARE statement.

Related reference:

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

Example of Parameter Markers in a Dynamic SQL Program

The following examples show how to use parameter markers in a dynamic SQL program:

- C/C++ (**dbuse.sqc/dbuse.sqC**)

  The function `DynamicStmtWithMarkersEXECUTEusingHostVars()` in the C-language sample **dbuse.sqc** shows how to perform a delete using a parameter marker with a host variable:

```
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 (**DbUse.java**)

  The function `execPreparedStatementWithParam()` in the JDBC sample **DbUse.java** shows how to perform a delete using parameter markers:

```
// prepare the statement with parameter markers
PreparedStatement prepStmt = con.prepareStatement(
  " DELETE FROM org WHERE deptnumb <= ? AND division = ? ");

// execute the statement
prepStmt.setInt(1, 70);
prepStmt.setString(2, "Eastern");
prepStmt.execute();

// close the statement
prepStmt.close();
```

- COBOL (**varinp.sqb**)

  The following example is from the COBOL sample **varinp.sqb**, and shows how to use a parameter marker in search and update conditions:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 pname             pic x(10).
01 dept              pic s9(4) comp-5.
01 st                pic x(127).
01 parm-var          pic x(5).
   EXEC SQL END DECLARE SECTION END-EXEC.

   move "SELECT name, dept FROM staff
-         "   WHERE job = ? FOR UPDATE OF job" to st.
   EXEC SQL PREPARE s1 FROM :st END-EXEC.
```

```
           EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

           move "Mgr" to parm-var.
           EXEC SQL OPEN c1 USING :parm-var END-EXEC

           move "Clerk" to parm-var.
           move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.
           EXEC SQL PREPARE s2 from :st END-EXEC.

       * call the FETCH and UPDATE loop.
           perform Fetch-Loop thru End-Fetch-Loop
              until SQLCODE not equal 0.

           EXEC SQL CLOSE c1 END-EXEC.
```

Related concepts:

- "Error Message Retrieval in an Application" on page 102

Related samples:

- "dbuse.out -- HOW TO USE A DATABASE (C)"
- "dbuse.sqc -- How to use a database (C)"
- "dbuse.out -- HOW TO USE A DATABASE (C++)"
- "dbuse.sqC -- How to use a database (C++)"
- "DbUse.java -- How to use a database (JDBC)"
- "DbUse.out -- HOW TO USE A DATABASE. Connect to 'sample' database using JDBC type 2 driver (JDBC)"

DB2 Call Level Interface (CLI) Compared to Dynamic SQL

The sections that follow describe the differences between DB2 CLI and dynamic SQL, the advantages that DB2 CLI has over dynamic SQL, and when you should use DB2 CLI or dynamic SQL.

DB2 Call Level Interface (CLI) versus embedded dynamic SQL

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the database, and executed. In contrast, a DB2 CLI application does not have to be precompiled or bound, but instead uses a standard set of functions to execute SQL statements and related services at run time.

This difference is important because, traditionally, precompilers have been specific to each database product, which effectively ties your applications to that product. DB2 CLI enables you to write portable applications that are independent of any particular database product. This independence means DB2 CLI applications do not have to be recompiled or rebound to access different DB2® databases, including host system databases. They just connect to the appropriate database at run time.

The following are differences and similarities between DB2 CLI and embedded SQL:

- DB2 CLI does not require the explicit declaration of cursors. DB2 CLI has a supply of cursors that get used as needed. The application can then use the generated cursor in the normal cursor fetch model for multiple row `SELECT` statements and positioned `UPDATE` and `DELETE` statements.

- The `OPEN` statement is not used in DB2 CLI. Instead, the execution of a `SELECT` automatically causes a cursor to be opened.
- Unlike embedded SQL, DB2 CLI allows the use of parameter markers on the equivalent of the `EXECUTE IMMEDIATE` statement (the `SQLExecDirect()` function).
- A `COMMIT` or `ROLLBACK` in DB2 CLI is typically issued via the `SQLEndTran()` function call rather than by executing it as an SQL statement, however, doing do is permitted.
- DB2 CLI manages statement related information on behalf of the application, and provides an abstract object to represent the information called a *statement handle*. This handle eliminates the need for the application to use product specific data structures.
- Similar to the statement handle, the *environment handle* and *connection handle* provide a means to refer to global variables and connection specific information. The *descriptor handle* describes either the parameters of an SQL statement or the columns of a result set.
- DB2 CLI applications can dynamically describe parameters in an SQL statement the same way that CLI and embedded SQL applications describe result sets. This enables CLI applications to dynamically process SQL statements that contain parameter markers without knowing the data type of those parameter markers in advance. When the SQL statement is prepared, describe information is returned detailing the data types of the parameters.
- DB2 CLI uses the SQLSTATE values defined by the X/Open SQL CAE specification. Although the format and most of the values are consistent with values used by the IBM® relational database products, there are differences. (There are also differences between ODBC SQLSTATES and the X/Open defined SQLSTATES).

Despite these differences, there is an important common concept between embedded SQL and DB2 CLI: *DB2 CLI can execute any SQL statement that can be prepared dynamically in embedded SQL.*

Note: DB2 CLI can also accept some SQL statements that cannot be prepared dynamically, such as compound SQL statements.

Each DBMS may have additional statements that you can dynamically prepare. In this case, DB2 CLI passes the statements directly to the DBMS. There is one exception: the COMMIT and ROLLBACK statements can be dynamically prepared by some DBMSs but will be intercepted by DB2 CLI and treated as an appropriate `SQLEndTran()` request. However, it is recommended you use the `SQLEndTran()` function to specify either the COMMIT or ROLLBACK statement.

Related reference:

- Appendix A, “Supported SQL Statements,” on page 685

Advantages of DB2 CLI over embedded SQL

The DB2 CLI interface has several key advantages over embedded SQL.

- It is ideally suited for a client-server environment, in which the target database is not known when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application is connected to.

- It increases the portability of applications by removing the dependence on precompilers. Applications are distributed not as embedded SQL source code which must be preprocessed for each database product, but as compiled applications or run time libraries.
- Individual DB2 CLI applications do not need to be bound to each database, only bind files shipped with DB2 CLI need to be bound once for all DB2 CLI applications. This can significantly reduce the amount of management required for the application once it is in general use.
- DB2 CLI applications can connect to multiple databases, including multiple connections to the same database, all from the same application. Each connection has its own commit scope. This is much simpler using CLI than using embedded SQL where the application must make use of multi-threading to achieve the same result.
- DB2 CLI eliminates the need for application controlled, often complex data areas, such as the SQLDA and SQLCA, typically associated with embedded SQL applications. Instead, DB2 CLI allocates and controls the necessary data structures, and provides a *handle* for the application to reference them.
- DB2 CLI enables the development of multi-threaded thread-safe applications where each thread can have its own connection and a separate commit scope from the rest. DB2 CLI achieves this by eliminating the data areas described above, and associating all such data structures that are accessible to the application with a specific handle. Unlike embedded SQL, a multi-threaded CLI application does not need to call any of the context management DB2® APIs; this is handled by the DB2 CLI driver automatically.
- DB2 CLI provides enhanced parameter input and fetching capability, allowing arrays of data to be specified on input, retrieving multiple rows of a result set directly into an array, and executing statements that generate multiple result sets.
- DB2 CLI provides a consistent interface to query catalog (Tables, Columns, Foreign Keys, Primary Keys, etc.) information contained in the various DBMS catalog tables. The result sets returned are consistent across DBMSs. This shields the application from catalog changes across releases of database servers, as well as catalog differences amongst different database servers; thereby saving applications from writing version specific and server specific catalog queries.
- Extended data conversion is also provided by DB2 CLI, requiring less application code when converting information between various SQL and C data types.
- DB2 CLI incorporates both the ODBC and X/Open CLI functions, both of which are accepted industry specifications. DB2 CLI is also aligned with the ISO CLI standard. Knowledge that application developers invest in these specifications can be applied directly to DB2 CLI development, and vice versa. This interface is intuitive to grasp for those programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.
- DB2 CLI provides the ability to retrieve multiple rows and result sets generated from a stored procedure residing on a DB2 Universal Database (or DB2 Universal Database for z/OS and OS/390 version 5 or later) server. However, note that this capability exists for Version 5 DB2 Universal Database clients using embedded SQL if the stored procedure resides on a server accessible from a DataJoiner® Version 2 server.
- DB2 CLI offers more extensive support for scrollable cursors. With scrollable cursors, you can scroll through a cursor as follows:
  - Forward by one or more rows

– Backward by one or more rows
– From the first row by one or more rows
– From the last row by one or more rows.

Scrollable cursors can be used in conjunction with array output. You can declare an updatable cursor as scrollable then move forward or backward through the result set by one or more rows. You can also fetch rows by specifying an offset from:

– The current row
– The beginning or end of the result set
– A specific row you have previously set with a bookmark.

When to use DB2 CLI or embedded SQL

Which interface you choose depends on your application.

DB2 CLI is ideally suited for query-based graphical user interface (GUI) applications that require portability. The advantages listed above, may make using DB2 CLI seem like the obvious choice for any application. There is however, one factor that must be considered, the comparison between static and dynamic SQL. It is much easier to use static SQL in embedded applications.

Static SQL has several advantages:

- Performance

  Dynamic SQL is prepared at run time, static SQL is prepared at precompile time. As well as requiring more processing, the preparation step may incur additional network-traffic at run time. The additional network traffic can be avoided if the DB2 CLI application makes use of deferred prepare (which is the default behavior).

  It is important to note that static SQL will not always have better performance than dynamic SQL. Dynamic SQL is prepared at runtime and uses the database statistics available at that time, whereas static SQL makes use of database statistics available at BIND time. Dynamic SQL can make use of changes to the database, such as new indexes, to choose the optimal access plan, resulting in potentially better performance than the same SQL executed as static SQL. In addition, precompilation of dynamic SQL statements can be avoided if they are cached.
- Encapsulation and Security

  In static SQL, the authorizations to access objects (such as a table, view) are associated with a package and are validated at package binding time. This means that database administrators need only to grant execute on a particular package to a set of users (thus encapsulating their privileges in the package) without having to grant them explicit access to each database object. In dynamic SQL, the authorizations are validated at run time on a per statement basis; therefore, users must be granted explicit access to each database object. This permits these users access to parts of the object that they do not have a need to access.
- Embedded SQL is supported in languages other than C or C++.
- For fixed query selects, embedded SQL is simpler.

If an application requires the advantages of both interfaces, it is possible to make use of static SQL within a DB2 CLI application by creating a stored procedure that contains the static SQL. The stored procedure is called from within a DB2 CLI

application and is executed on the server. Once the stored procedure is created, any DB2 CLI or ODBC application can call it.

It is also possible to write a mixed application that uses both DB2 CLI and embedded SQL, taking advantage of their respective benefits. In this case, DB2 CLI is used to provide the base application, with key modules written using static SQL for performance or security reasons. This complicates the application design, and should only be used if stored procedures do not meet the applications requirements.

Ultimately, the decision on when to use each interface, will be based on individual preferences and previous experience rather than on any one factor.

Related concepts:

- "CLI/ODBC/JDBC trace facility" on page 460

Related tasks:

- "Preparing and executing SQL statements in CLI applications" in the *CLI Guide and Reference, Volume 1*
- "Issuing SQL statements in CLI applications" in the *CLI Guide and Reference, Volume 1*
- "Creating static SQL with CLI/ODBC/JDBC Static Profiling" in the *CLI Guide and Reference, Volume 1*

Chapter 6. Programming in C and C++

Programming Considerations for C/C++

Special host language programming considerations are discussed in the following topics. Included is information on language restrictions, host-language-specific include files, embedding SQL statements, host variables, and supported data types for host variables.

Related reference:

- "C samples" in the *Application Development Guide: Building and Running Applications*

Trigraph Sequences for C and C++

Some characters from the C or C++ character set are not available on all keyboards. These characters can be entered into a C or C++ source program using a sequence of three characters called a *trigraph*. Trigraphs are not recognized in SQL statements. The precompiler recognizes the following trigraphs within host variable declarations:

Trigraph	Definition
??(	Left bracket '['
??)	Right bracket ']'

??<	Left brace '{'
??>	Right brace '}'

The remaining trigraphs listed below may occur elsewhere in a C or C++ source program:

Trigraph	Definition
??=	Hash mark '#'
??/	Back slash '\'
??'	Caret '^'
??!	Vertical Bar '|'
??–	Tilde '~'

Input and Output Files for C and C++

By default, the input file can have the following extensions:

.sqc For C files on all supported platforms

.sqC For C++ files on UNIX® platforms

.sqx For C++ files on Windows® operating systems

By default, the corresponding precompiler output files have the following extensions:

.c For C files on all supported platforms

.C For C++ files on UNIX platforms

.cxx For C++ files on Windows operating systems

You can use the OUTPUT precompile option to override the name and path of the output modified source file. If you use the TARGET C or TARGET CPLUSPLUS precompile option, the input file does not need a particular extension.

Include Files

The following sections describe include files for C and C++.

Include Files for C and C++

The host-language-specific include files (header files) for C and C++ have the file extension `.h`. The include files that are intended to be used in your applications are described below.

SQL (sql.h)
This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLADEF (sqladef.h)
This file contains function prototypes used by precompiled C and C++ applications.

SQLAPREP (sqlaprep.h)
This file contains definitions required to write your own precompiler.

SQLCA (sqlca.h)
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

SQLCLI (sqlcli.h)
This file contains the function prototypes and constants needed to write a Call Level Interface (DB2 CLI) application. The functions in this file are common to both X/Open Call Level Interface and ODBC Core Level.

SQLCLI1 (sqlcli1.h)
This file contains the function prototypes and constants needed to write a Call Level Interface (DB2 CLI) that makes use of the more advanced features in DB2 CLI. Many of the functions in this file are common to both X/Open Call Level Interface and ODBC Level 1. In addition, this file also includes X/Open-only functions and DB2-specific functions.

This file includes both `sqlcli.h` and `sqlext.h` (which contains ODBC Level2 API definitions).

SQLCODES (sqlcodes.h)
This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqlda.h)
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

SQLENV (sqlenv.h)
This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLEXT (sqlext.h)
This file contains the function prototypes and constants of those ODBC Level 1 and Level 2 APIs that are not part of the X/Open Call Level Interface specification and is therefore used with the permission of Microsoft Corporation.

SQLE819A (sqle819a.h)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.h)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.h)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.h)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.h)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.h)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLJACB (sqljacb.h)
This file defines constants, structures, and control blocks for the DB2 Connect interface.

SQLMON (sqlmon.h)
This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLSTATE (sqlstate.h)
This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLSYSTM (sqlsystm.h)
This file contains the platform-specific definitions used by the database manager APIs and data structures.

SQLUDF (sqludf.h)
This file defines constants and interface structures for writing user-defined functions (UDFs).

SQLUTIL (sqlutil.h)
This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

SQLUV (sqluv.h)
This file defines structures, constants, and prototypes for the asynchronous Read Log API, and APIs used by the table load and unload vendors.

SQLUVEND (sqluvend.h)
This file defines structures, constants, and prototypes for the APIs to be used by the storage management vendors.

SQLXA (sqlxa.h)
This file contains function prototypes and constants used by applications that use the X/Open XA Interface.

Related concepts:

• "Include Files in C and C++" on page 134

Include Files in C and C++

There are two methods for including files: the EXEC SQL INCLUDE statement and the #include macro. The precompiler will ignore the #include, and only process files included with the EXEC SQL INCLUDE statement.

To locate files included using `EXEC SQL INCLUDE`, the DB2® C precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll;`

If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the C precompiler searches for `payroll.sqc`, then `payroll.h`, in each directory in which it looks. On UNIX® operating systems, the C++ precompiler searches for `payroll.sqC`, then `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory in which it looks. On Windows®-32 bit operating systems, the C++ precompiler searches for `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory in which it looks.

- `EXEC SQL INCLUDE 'pay/payroll.h';`

  If the file name is enclosed in quotation marks, as above, no extension is added to the name.

  If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, on UNIX-based systems, if DB2INCLUDE is set to '`/disk2:myfiles/c`', the C/C++ precompiler searches for '`./pay/payroll.h`', then '`/disk2/pay/payroll.h`', and finally '`./myfiles/c/pay/payroll.h`'. The path where the file is actually found is displayed in the precompiler messages. On Windows-based operating systems, substitute back slashes (\) for the forward slashes in the above example.

Note: The setting of DB2INCLUDE is cached by the command line processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

To help relate compiler errors back to the original source the precompiler generates ANSI #line macros in the output file. This allows the compiler to report errors using the file name and line number of the source or included source file, rather than the precompiler output.

However, if you specify the PREPROCESSOR option, all the #line macros generated by the precompiler reference the preprocessed file from the external C preprocessor.

Some debuggers and other tools that relate source code to object code do not always work well with the #line macro. If the tool you want to use behaves unexpectedly, use the NOLINEMACRO option (used with DB2 PREP) when precompiling. This option prevents the #line macros from being generated.

Related concepts:

- "C Macro Expansion" on page 149

Related reference:

- "PREPARE statement" in the *SQL Reference, Volume 2*
- "Include Files for C and C++" on page 132

Embedded SQL Statements in C and C++

Embedded SQL statements consist of the following three elements:

Element	Correct Syntax
Statement initializer	EXEC SQL
Statement string	Any valid SQL statement
Statement terminator	semicolon (;)

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

The following rules apply to embedded SQL statements:

- You can begin the SQL statement string on the same line as the keyword pair or a separate line. The statement string can be several lines long. Do not split the EXEC SQL keyword pair between lines.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.

  C/C++ comments can be placed before the statement initializer or after the statement terminator.
- Multiple SQL statements and C/C++ statements may be placed on the same line. For example:

  ```
  EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
  ```
- The SQL precompiler leaves carriage returns, line feeds, and TABs in a quoted string as is.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (`--`) followed by a string of zero or more characters, and terminated by a line end. Do not place SQL comments after the SQL statement terminator. Comments after the terminator cause compilation errors because they appear to be part of the C/C++ language.

  You can use comments in a static statement string wherever blanks are allowed. Use the C/C++ comment delimiters `/* */`, or the SQL comment symbol (`--`). //-style C++ comments are not permitted within static SQL statements, but they may be used elsewhere in your program. The precompiler removes comments before processing the SQL statement. You **cannot** use the C and C++ comment delimiters `/* */` or `//` in a dynamic SQL statement. However, you can use them elsewhere in your program.
- You can continue SQL string literals and delimited identifiers over line breaks in C and C++ applications. To do this, use a back slash (\) at the end of the line where the break is desired. For example:

  ```
  EXEC SQL SELECT "NA\
  ME" INTO :n FROM staff WHERE name='Sa\
  nders';
  ```

  Any new line characters (such as carriage return and line feed) are not included in the string or delimited identifier.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a C program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, UNIX®-based systems use a line feed.

Related reference:

- Appendix A, “Supported SQL Statements,” on page 685

Host Variables in C and C++

The sections that follow describe how to declare and use host variables in C and C++ programs.

Host Variables in C and C++

Host variables are C or C++ language variables that are referenced within SQL statements. They allow an application to pass input data to and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other C/C++ variable. Follow the rules described in the following sections when naming, declaring, and using host variables.

In applications that manually construct the SQLDA, long variables cannot be used when sqlvar::sqltype==SQL_TYP_INTEGER. Instead, sqlint32 types must be used. This problem is identical to using long variables in host variable declarations, except that with a manually constructed SQLDA, the precompiler will not uncover this error and run time errors will occur.

Any long and unsigned long casts that are used to access sqlvar::sqldata information must be changed to sqlint32 and sqluint32. Val members for the sqloptions and sqla_option structures are declared as sqluintptr. Therefore, assignment of pointer members into sqla_option::val or sqloptions::val members should use sqluintptr casts rather than unsigned long casts. This change will not cause run-time problems in 64-bit UNIX® platforms, but should be made in preparation for 64-bit Windows® NT applications, where the long type is only 32-bit.

Related concepts:

- "Host Variable Names in C and C++" on page 137
- "Host Variable Declarations in C and C++" on page 138
- "Syntax for Fixed and Null-Terminated Character Host Variables in C and C++" on page 140
- "Indicator Variables in C and C++" on page 142
- "Graphic Host Variables in C and C++" on page 143
- "Host Variable Initialization in C and C++" on page 149
- "Host Structure Support in C and C++" on page 150
- "SQL Declare Section with Host Variables for C and C++" on page 161

Related reference:

- "Syntax for Numeric Host Variables in C and C++" on page 139
- "Syntax for Variable-Length Character Host Variables in C or C++" on page 141
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 146
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 147
- "Syntax for File Reference Host Variable Declarations in C or C++" on page 148

Host Variable Names in C and C++

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than SQL, sql, DB2®, and db2, which are reserved for system use. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* allowed */
  char  sqlvar;     /* not allowed */
  char  SQL_VAR;    /* not allowed */
EXEC SQL END DECLARE SECTION;
```

- The precompiler considers host variable names as global to a module. This does not mean, however, that host variables have to be declared as global variables; it is perfectly acceptable to declare host variables as local variables within functions. For example, the following code will work correctly:

```
void f1(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_1;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
}
void f2(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
}
```

  It is also possible to have several local host variables with the same name, as long as they all have the same type and size. To do this, declare the first occurrence of the host variable to the precompiler between BEGIN DECLARE SECTION and END DECLARE SECTION statements, and leave subsequent declarations of the variable out of declare sections. The following code shows an example of this:

```
void f3(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  char host_var_3[25];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
}
void f4(int i)
{
char host_var_3[25];
EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
}
```

  Because f3 and f4 are in the same module, and host_var_3 has the same type and length in both functions, a single declaration to the precompiler is sufficient to use it in both places.

Related concepts:

- "Host Variable Declarations in C and C++" on page 138

Host Variable Declarations in C and C++

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The C/C++ precompiler only recognizes a subset of valid C or C++ declarations as valid host variable declarations. These declarations define either numeric or

character variables. Typedefs for host variable types are not allowed. Host variables can be grouped into a single host structure. You can declare C++ class data members as host variables.

A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time, or timestamp SQL input or output value. The application must ensure that output variables are long enough to contain the values that they receive.

Related concepts:

- "Syntax for Fixed and Null-Terminated Character Host Variables in C and C++" on page 140
- "Graphic Host Variables in C and C++" on page 143
- "Host Structure Support in C and C++" on page 150
- "Class Data Members Used as Host Variables in C and C++" on page 155

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 29
- "Declaring structured type host variables" in the *Application Development Guide: Programming Server Applications*

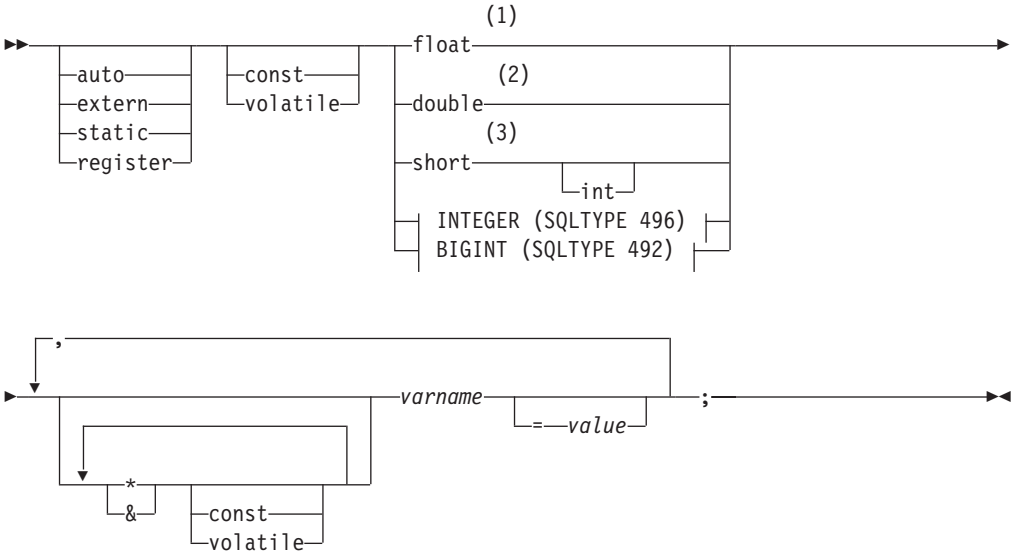
Related reference:

- "Syntax for Numeric Host Variables in C and C++" on page 139
- "Syntax for Variable-Length Character Host Variables in C or C++" on page 141

Syntax for Numeric Host Variables in C and C++

Following is the syntax for declaring numeric host variables in C or C++.

Syntax for Numeric Host Variables in C or C++



INTEGER (SQLTYPE 496)

BIGINT (SQLTYPE 492)

Notes:

1 REAL (SQLTYPE 480), length 4

2 DOUBLE (SQLTYPE 480), length 8

3 SMALLINT (SQLTYPE 500)

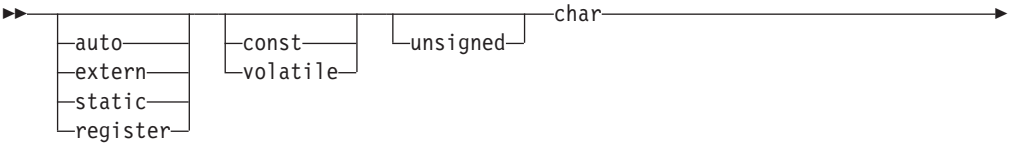
4 For maximum application portability, use sqlint32 and sqlint64 for INTEGER and BIGINT host variables, respectively. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

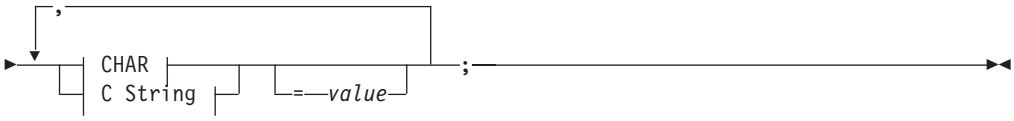
5 For maximum application portability, use sqlint32 and sqlint64 for INTEGER and BIGINT host variables, respectively. To use the BIGINT data type, your platform must support 64 bit integer values. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

Syntax for Fixed and Null-Terminated Character Host Variables in C and C++

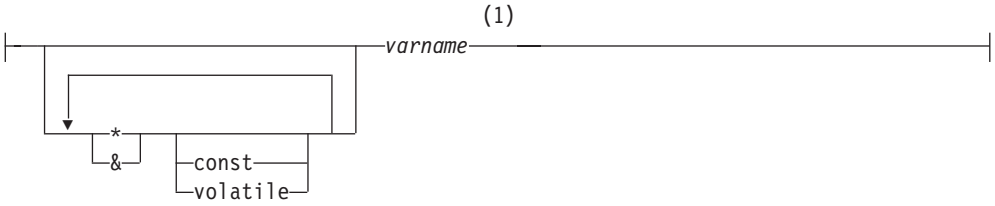
Following is the syntax for declaring fixed and null-terminated character host variables in C or C++.

Syntax for Fixed and Null-Terminated Character Host Variables

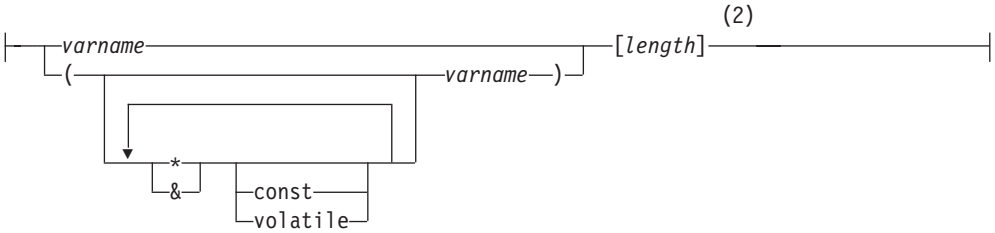




CHAR



C String



Notes:

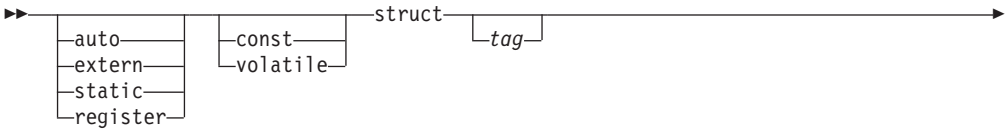
1 CHAR (SQLTYPE 452), length 1

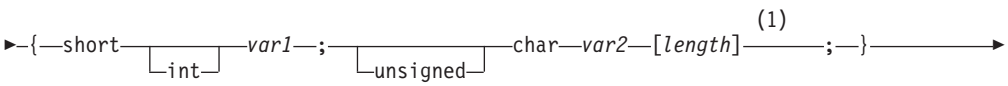
2 Null-terminated C string (SQLTYPE 460); length can be any valid constant expression

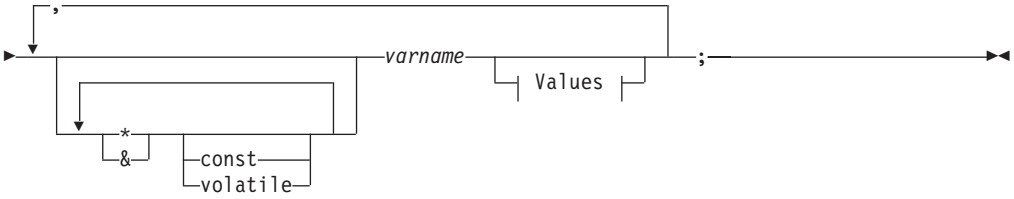
Syntax for Variable-Length Character Host Variables in C or C++

Following is the syntax for declaring variable-length character host variables in C or C++.

Syntax for Variable-Length Character Host Variables in C







Values

Notes:

1 In form 2, length can be any valid constant expression. Its value after evaluation determines if the host variable is VARCHAR (SQLTYPE 448) or LONG VARCHAR (SQLTYPE 456).

Variable-Length Character Host Variable Considerations:

1. Although the database manager converts character data to either **form 1** or **form 2** whenever possible, **form 1** corresponds to column types CHAR or VARCHAR, while **form 2** corresponds to column types VARCHAR and LONG VARCHAR.
2. If **form 1** is used with a length specifier *[n]*, the value for the length specifier after evaluation must be no greater than 32 672, and the string contained by the variable should be null-terminated.
3. If **form 2** is used, the value for the length specifier after evaluation must be no greater than 32 700.
4. In **form 2**, *var1* and *var2* must be simple variable references (no operators), and cannot be used as host variables (*varname* is the host variable).
5. *varname* can be a simple variable name, or it can include operators such as **varname*. See the description of pointer data types in C and C++ for more information.
6. The precompiler determines the SQLTYPE and SQLLEN of all host variables. If a host variable appears in an SQL statement with an indicator variable, the SQLTYPE is assigned to be the base SQLTYPE plus one for the duration of that statement.
7. The precompiler permits some declarations which are not syntactically valid in C or C++. Refer to your compiler documentation if in doubt about a particular declaration syntax.

Related concepts:

• "Host Variables Used as Pointer Data Types in C and C++" on page 154

Indicator Variables in C and C++

Indicator variables should be declared as a `short` data type.

Related concepts:

• "Indicator Tables in C and C++" on page 152

Graphic Host Variables in C and C++

To handle graphic data in C or C++ applications, use host variables based on either the `wchar_t` C/C++ data type or the `sqldbchar` data type provided by DB2®. You can assign these types of host variables to columns of a table that are GRAPHIC, VARGRAPHIC, or DBCLOB. For example, you can update or select DBCS data from GRAPHIC or VARGRAPHIC columns of a table.

There are three valid forms for a graphic host variable:

- Single-graphic form

  Single-graphic host variables have an SQLTYPE of 468/469 that is equivalent to the GRAPHIC(1) SQL data type.
- Null-terminated graphic form

  Null-terminated refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). They have an SQLTYPE of 400®/401.
- VARGRAPHIC structured form

  VARGRAPHIC structured host variables have an SQLTYPE of 464/465 if their length is between 1 and 16 336 bytes. They have an SQLTYPE of 472/473 if their length is between 2 000 and 16 350 bytes.

Related concepts:

- "Host Variable Names in C and C++" on page 137
- "Host Variable Declarations in C and C++" on page 138
- "Host Variable Initialization in C and C++" on page 149
- "Host Structure Support in C and C++" on page 150
- "Indicator Tables in C and C++" on page 152
- "Multi-Byte Character Encoding in C and C++" on page 156
- "wchar_t and sqldbchar Data Types in C and C++" on page 157
- "WCHARTYPE Precompiler Option in C and C++" on page 158

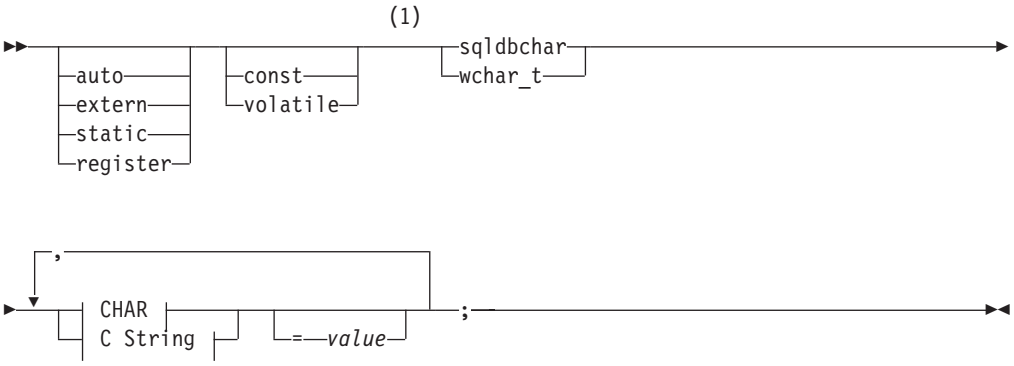
Related reference:

- "Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++" on page 143
- "Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++" on page 145
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 146
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 147
- "Syntax for File Reference Host Variable Declarations in C or C++" on page 148

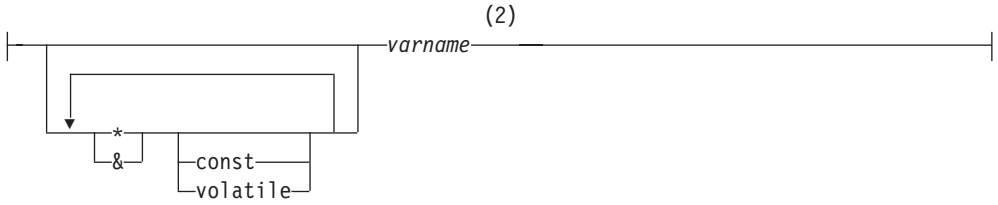
Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++

Following is the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.

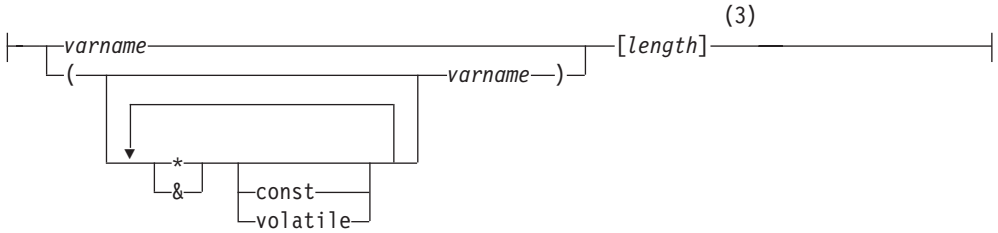
Syntax for Graphic Declaration of Single-Graphic Form and



CHAR



C String



Notes:

1 To determine which of the two graphic types should be used, see the description of the wchar_t and sqldbchar data types in C and C++.

2 GRAPHIC (SQLTYPE 468), length 1

3 Null-terminated graphic string (SQLTYPE 400)

Graphic Host Variable Considerations:

1. The single-graphic form declares a fixed-length graphic string host variable of length 1 with SQLTYPE of 468 or 469.
2. *value* is an initializer. A wide-character string literal (L-literal) should be used if the WCHARTYPE CONVERT precompiler option is used.
3. *length* can be any valid constant expression, and its value after evaluation must be greater than or equal to 1, and not greater than the maximum length of VARGRAPHIC, which is 16 336.
4. Null-terminated graphic strings are handled differently, depending on the value of the standards level precompile option setting.

Related concepts:

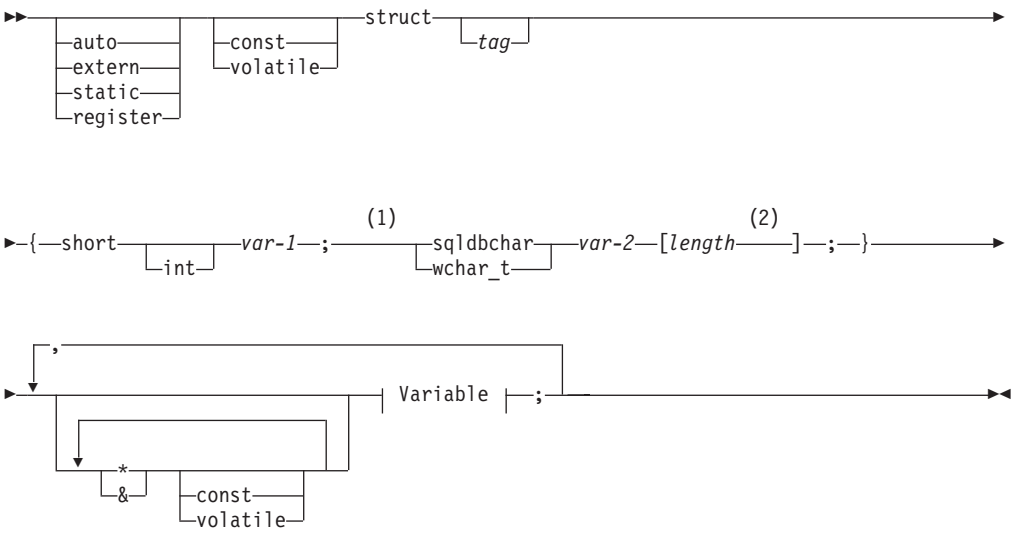
- "Null-Terminated Strings in C and C++" on page 153

- "wchar_t and sqldbchar Data Types in C and C++" on page 157

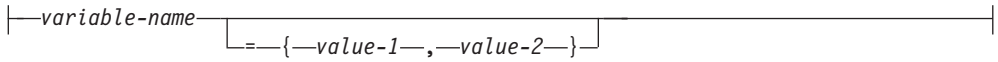
Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++

Following is the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

Syntax for Graphic Declaration of VARGRAPHIC Structured



Variable:



Notes:

1 To determine which of the two graphic types should be used, see the description of the wchar_t and sqldbchar data types in C and C++.

2 *length* can be any valid constant expression. Its value after evaluation determines if the host variable is VARGRAPHIC (SQLTYPE 464) or LONG VARGRAPHIC (SQLTYPE 472). The value of length must be greater than or equal to 1, and not greater than the maximum length of LONG VARGRAPHIC which is 16 350.

Graphic Declaration (VARGRAPHIC Structured Form) Considerations:

1. *var-1* and *var-2* must be simple variable references (no operators) and cannot be used as host variables.
2. *value-1* and *value-2* are initializers for *var-1* and *var-2*. *value-1* must be an integer and *value-2* should be a wide-character string literal (L-literal) if the WCHARTYPE CONVERT precompiler option is used.
3. The struct *tag* can be used to define other data areas, but itself cannot be used as a host variable.

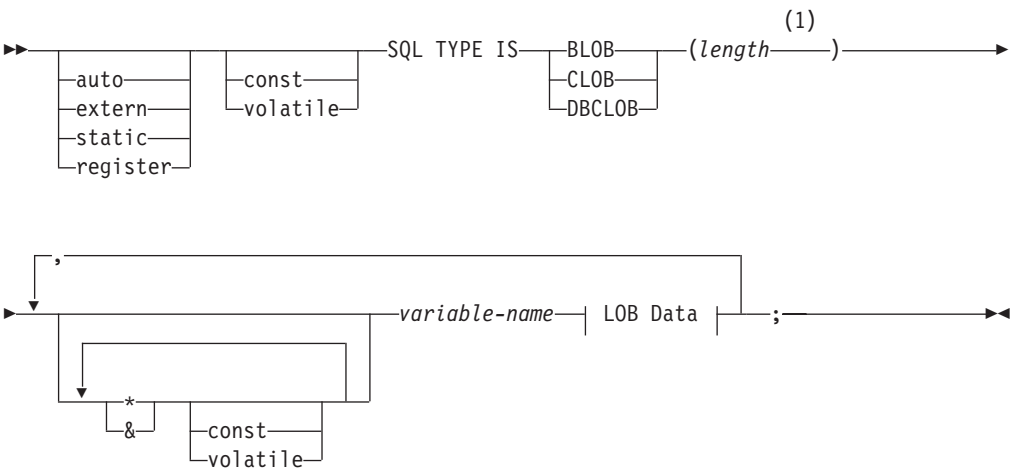
Related concepts:

- "wchar_t and sqldbchar Data Types in C and C++" on page 157

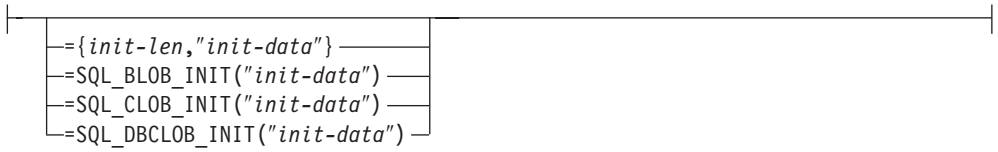
Syntax for Large Object (LOB) Host Variables in C or C++

Following is the syntax for declaring large object (LOB) host variables in C or C++.

Syntax for Large Object (LOB) Host Variables in C or C++



LOB Data



Notes:

1 *length* can be any valid constant expression, in which the constant K, M, or G can be used. The value of length after evaluation for BLOB and CLOB must be 1 <= length <= 2 147 483 647. The value of *length* after evaluation for DBCLOB must be 1 <= length <= 1 073 741 823.

LOB Host Variable Considerations:

1. The SQL TYPE IS clause is needed to distinguish the three LOB-types from each other so that type checking and function resolution can be carried out for LOB-type host variables that are passed to functions.
2. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G may be in mixed case.
3. The maximum length allowed for the initialization string *"init-data"* is 32 702 bytes, including string delimiters (the same as the existing limit on C/C++ strings within the precompiler).
4. The initialization length, *init-len*, must be a numeric constant (i.e. it cannot include K, M, or G).
5. A length for the LOB must be specified; that is, the following declaration is not permitted:

   ```
   SQL TYPE IS BLOB my_blob;
   ```

6. If the LOB is not initialized within the declaration, no initialization will be done within the precompiler-generated code.
7. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

Note: Wide-character literals, for example, `L"Hello"`, should only be used in a precompiled program if the WCHARTYPE CONVERT precompile option is selected.

8. The precompiler generates a structure tag which can be used to cast to the host variable's type.

BLOB Example:

Declaration:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

Results in the generation of the following structure:

```
static struct my_blob_t {
      sqluint32        length;
      char             data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

CLOB Example:

Declaration:

```
 volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

Results in the generation of the following structure:

```
 volatile struct var1_t {
      sqluint32        length;
      char             data[131072000];
} * var1, var2 = {10, "data5data5"};
```

DBCLOB Example:

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

Precompiled with the WCHARTYPE NOCONVERT option, results in the generation of the following structure:

```
struct my_dbclob1_t {
     sqluint32        length;
     sqldbchar        data[30000];
} my_dbclob1;
```

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

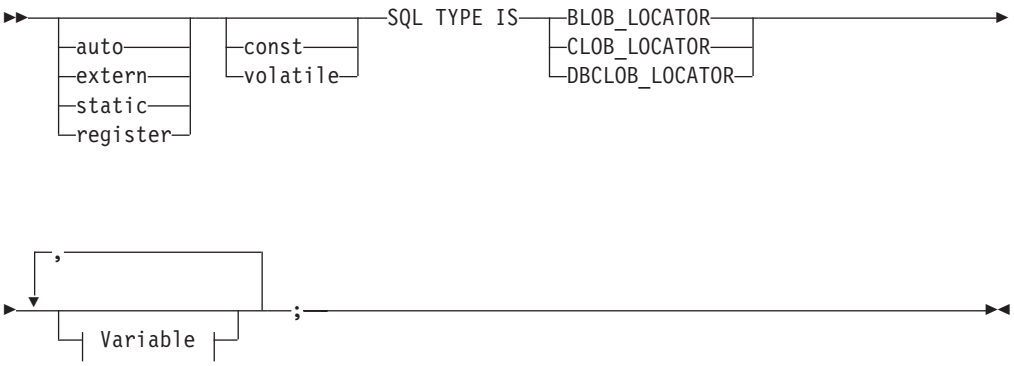
Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

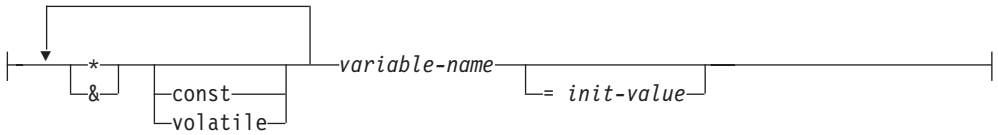
Syntax for Large Object (LOB) Locator Host Variables in C or C++

Following is the syntax for declaring large object (LOB) locator host variables in C or C++.

Syntax for Large Object (LOB) Locator Host Variables in



Variable



LOB Locator Host Variable Considerations:

1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR may be in mixed case.
2. *init-value* permits the initialization of pointer and reference locator variables. Other types of initialization will have no meaning.

CLOB Locator Example (other LOB locator type declarations are similar):

Declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

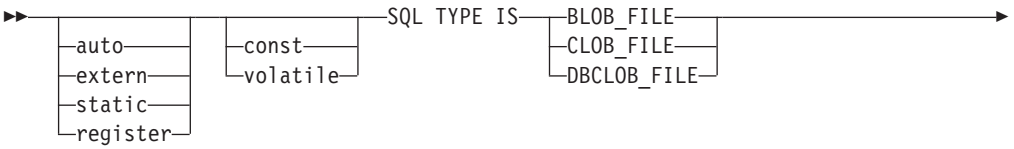
Results in the generation of the following declaration:

```
sqluint32 my_locator;
```

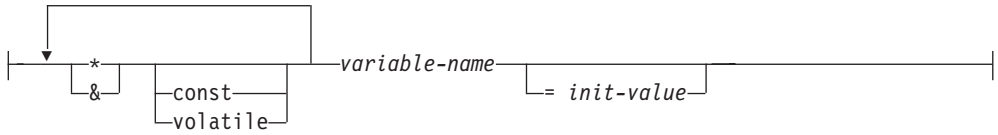
Syntax for File Reference Host Variable Declarations in C or C++

Following is the syntax for declaring file reference host variables in C or C++.

Syntax for File Reference Host Variables in C or C++



Variable



Note: SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE may be in mixed case.

CLOB File Reference Example (other LOB file reference type declarations are similar):

Declaration:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

Results in the generation of the following structure:

```
static volatile struct {
     sqluint32        name_length;
     sqluint32        data_length;
     sqluint32        file_options;
              char    name[255];
} my_file;
```

Host Variable Initialization in C and C++

In C++ declare sections, you cannot initialize host variables using parentheses. The following example shows the correct and incorrect methods of initialization in a declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
  short my_short_2 = 5;       /* correct   */
  short my_short_1(5);        /* incorrect */
EXEC SQL END DECLARE SECTION;
```

C Macro Expansion

The C/C++ precompiler cannot directly process any C macro used in a declaration within a declare section. Instead, you must first preprocess the source file with an external C preprocessor. To do this, specify the exact command for invoking a C preprocessor to the precompiler through the PREPROCESSOR option.

When you specify the PREPROCESSOR option, the precompiler first processes all the SQL INCLUDE statements by incorporating the contents of all the files referred to in the SQL INCLUDE statement into the source file. The precompiler then invokes the external C preprocessor using the command you specify with the modified source file as input. The preprocessed file, which the precompiler always expects to have an extension of `.i`, is used as the new source file for the rest of the precompiling process.

Any #line macro generated by the precompiler no longer references the original source file, but instead references the preprocessed file. To relate any compiler errors back to the original source file, retain comments in the preprocessed file. This helps you to locate various sections of the original source files, including the header files. The option to retain comments is commonly available in C preprocessors, and you can include the option in the command you specify through the PREPROCESSOR option. You should not have the C preprocessor output any #line macros itself, as they may be incorrectly mixed with ones generated by the precompiler.

Notes on Using Macro Expansion:

1. The command you specify through the PREPROCESSOR option should include all the desired options, but not the name of the input file. For example, for IBM® C on AIX® you can use the option:
   ```
   xlC -P -DMYMACRO=1
   ```
2. The precompiler expects the command to generate a preprocessed file with a `.i` extension. However, you cannot use redirection to generate the preprocessed file. For example, you **cannot** use the following option to generate a preprocessed file:
   ```
   xlC -E > x.i
   ```
3. Any errors the external C preprocessor encounters are reported in a file with a name corresponding to the original source file, but with a `.err` extension.

For example, you can use macro expansion in your source code as follows:

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
  struct
  {
    short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

The previous declarations resolve to the following after you use the PREPROCESSOR option:

```
EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
  struct
  {
    short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

Host Structure Support in C and C++

With host structure support, the C/C++ precompiler allows host variables to be grouped into a single host structure. This feature provides a shorthand for

referencing that same set of host variables in an SQL statement. For example, the following host structure can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
       short   years;
       double salary;
     } info;
   } staff_record;
```

The fields of a host structure can be any of the valid host variable types. Valid types include all numeric, character, and large object types. Nested host structures are also supported up to 25 levels. In the example above, the field `info` is a sub-structure, whereas the field `name` is not, as it represents a VARCHAR field. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. Pointer to host structure is also supported.

There are two ways to reference the host variables grouped in a host structure in an SQL statement:

- The host structure name can be referenced in an SQL statement.

  ```
  EXEC SQL SELECT id, name, years, salary
       INTO :staff_record
       FROM staff
       WHERE id = 10;
  ```

  The precompiler converts the reference to `staff_record` into a list, separated by commas, of all the fields declared within the host structure. Each field is qualified with the host structure names of all levels to prevent naming conflicts with other host variables or fields. This is equivalent to the following method.
- Fully qualified host variable names can be referenced in an SQL statement.

  ```
  EXEC SQL SELECT id, name, years, salary
       INTO :staff_record.id, :staff_record.name,
            :staff_record.info.years, :staff_record.info.salary
       FROM staff
       WHERE id = 10;
  ```

  References to field names must be fully qualified, even if there are no other host variables with the same name. Qualified sub-structures can also be referenced. In the example above, `:staff_record.info` can be used to replace `:staff_record.info.years, :staff_record.info.salary`.

Because a reference to a host structure (first example) is equivalent to a comma-separated list of its fields, there are instances where this type of reference may lead to an error. For example:

```
EXEC SQL DELETE FROM :staff_record;
```

Here, the DELETE statement expects a single character-based host variable. By giving a host structure instead, the statement results in a precompile-time error:

```
SQL0087N  Host variable "staff_record" is a structure used where structure
references are not permitted.
```

Other uses of host structures, which may cause an SQL0087N error to occur, include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables and SQLDA references. Host structures with exactly one field are permitted in such situations, as are references to individual fields (second example).

Related concepts:

- "Indicator Tables in C and C++" on page 152

Indicator Tables in C and C++

An indicator table is a collection of indicator variables to be used with a host structure. It must be declared as an array of short integers. For example:

```
short ind_tab[10];
```

The example above declares an indicator table with 10 elements. The following shows the way it can be used in an SQL statement:

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
     FROM staff
     WHERE id = 10;
```

The following lists each host structure field with its corresponding indicator variable in the table:

staff_record.id	ind_tab[0]
staff_record.name	ind_tab[1]
staff_record.info.years	ind_tab[2]
staff_record.info.salary	ind_tab[3]

Note: An indicator table element, for example `ind_tab[1]`, cannot be referenced individually in an SQL statement. The keyword INDICATOR is optional. The number of structure fields and indicators do not have to match; any extra indicators are unused, as are extra fields that do not have indicators assigned to them.

A scalar indicator variable can also be used in the place of an indicator table to provide an indicator for the first field of the host structure. This is equivalent to having an indicator table with only one element. For example:

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

If an indicator table is specified along with a host variable instead of a host structure, only the first element of the indicator table, for example `ind_tab[0]`, will be used:

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

If an array of short integers is declared within a host structure:

```
struct tag
{
  short i[2];
} test_record;
```

The array will be expanded into its elements when `test_record` is referenced in an SQL statement making `:test_record` equivalent to `:test_record.i[0]`, `:test_record.i[1]`.

Related concepts:

- "Host Structure Support in C and C++" on page 150

Null-Terminated Strings in C and C++

C/C++ null-terminated strings have their own SQLTYPE (460/461 for character and 468/469 for graphic).

C/C++ null-terminated strings are handled differently, depending on the value of the LANGLEVEL precompiler option. If a host variable of one of these SQLTYPE values and declared length *n* is specified within an SQL statement, and the number of bytes (for character types) or double-byte characters (for graphic types) of data is *k*, then:

- If the LANGLEVEL option on the PREP command is SAA1 (the default):

  For Output:

If...	Then...
k > *n*	*n* characters are moved to the target host variable, SQLWARN1 is set to 'W', and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to *k*.
k = *n*	*k* characters are moved to the target host variable, SQLWARN1 is set to 'N', and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.
k < *n*	*k* characters are moved to the target host variable and a null character is placed in character *k + 1*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

  For Input: When the database manager encounters an input host variable of one of these SQLTYPE values that does not end with a null-terminator, it will assume that character *n+1* will contain the null-terminator character.

- If the LANGLEVEL option on the PREP command is MIA:

  For Output:

If...	Then...

k >= *n*	*n - 1* characters are moved to the target host variable, SQLWARN1 is set to 'W', and SQLCODE 0 (SQLSTATE 01501). The *n*th character is set to the null-terminator. If an indicator variable was specified with the host variable, the value of the indicator variable is set to *k*.
k + 1 = *n*	*k* characters are moved to the target host variable, and the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.
k + 1 < *n*	*k* characters are moved to the target host variable, *n - k -1* blanks are appended on the right starting at character *k + 1*, then the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

For Input: When the database manager encounters an input host variable of one of these SQLTYPE values that does not end with a null character, SQLCODE -302 (SQLSTATE 22501) is returned.

When specified in any other SQL context, a host variable of SQLTYPE 460 with length *n* is treated as a VARCHAR data type with length *n*, as defined above. When specified in any other SQL context, a host variable of SQLTYPE 468 with length *n* is treated as a VARGRAPHIC data type with length *n*, as defined above.

Host Variables Used as Pointer Data Types in C and C++

Host variables may be declared as pointers to specific data types with the following restrictions:

- If a host variable is declared as a pointer, no other host variable may be declared with that same name within the same source file. The following example is not allowed:

```
char mystring[20];
char (*mystring)[20];
```

- Use parentheses when declaring a pointer to a null-terminated character array. In all other cases, parentheses are not allowed. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char  (*arr)[10];  /* correct   */
  char  *(arr);      /* incorrect */
  char  *arr[10];    /* incorrect */
EXEC SQL END DECLARE SECTION;
```

  The first declaration is a pointer to a 10-byte character array. This is a valid host variable. The second is an invalid declaration. The parentheses are not allowed in a pointer to a character. The third declaration is an array of pointers. This is not a supported data type.

  The host variable declaration:

```
char *ptr
```

  is accepted, but it does not mean *null-terminated character string of undetermined length*. Instead, it means a *pointer to a fixed-length, single-character host variable*.

This may not be what is intended. To define a pointer host variable that can indicate different character strings, use the first declaration form above.

- When pointer host variables are used in SQL statements, they should be prefixed by the same number of asterisks as they were declared with, as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char (*mychar)[20];    /* Pointer to character array of 20 bytes */
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT column INTO :*mychar FROM table;   /* Correct */
```

- Only the asterisk may be used as an operator over a host variable name.
- The maximum length of a host variable name is not affected by the number of asterisks specified, because asterisks are not considered part of the name.
- Whenever using a pointer variable in an SQL statement, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager.

Class Data Members Used as Host Variables in C and C++

You can declare class data members as host variables (but not classes or objects themselves). The following example illustrates the method to use:

```
class STAFF
{
    private:
         EXEC SQL BEGIN DECLARE SECTION;
           char        staff_name[20];
           short int   staff_id;
           double      staff_salary;
         EXEC SQL END DECLARE SECTION;
         short       staff_in_db;
    .
    .
};
```

Data members are only directly accessible in SQL statements through the implicit *this* pointer provided by the C++ compiler in class member functions. You **cannot** explicitly qualify an object instance (such as `SELECT name INTO :my_obj.staff_name ...`) in an SQL statement.

If you directly refer to class data members in SQL statements, the database manager resolves the reference using the *this* pointer. For this reason, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager. (This is true whenever pointer host variables are involved in SQL statements.)

The following example shows how you might directly use class data members which you have declared as host variables in an SQL statement.

```
class STAFF
{

⋮
    public:

⋮
       short int hire( void )
```

```
        {
          EXEC SQL INSERT INTO staff ( name,id,salary )
            VALUES ( :staff_name, :staff_id, :staff_salary );
          staff_in_db = (sqlca.sqlcode == 0);
          return sqlca.sqlcode;
        }
};
```

In this example, class data members `staff_name`, `staff_id`, and `staff_salary` are used directly in the INSERT statement. Because they have been declared as host variables (see the first example in this section), they are implicitly qualified to the current object with the *this* pointer. In SQL statements, you can also refer to data members that are not accessible through the *this* pointer. You do this by referring to them indirectly using pointer or reference host variables.

The following example shows a new method, *asWellPaidAs* that takes a second object, *otherGuy*. This method references its members indirectly through a local pointer or reference host variable, as you cannot reference its members directly within the SQL statement.

```
short int STAFF::asWellPaidAs( STAFF otherGuy )
{
    EXEC SQL BEGIN DECLARE SECTION;
      short &otherID = otherGuy.staff_id
      double otherSalary;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL SELECT SALARY INTO :otherSalary
      FROM STAFF WHERE id = :otherID;
      if( sqlca.sqlcode == 0 )
         return staff_salary >= otherSalary;
      else
         return 0;
}
```

Qualification and Member Operators in C and C++

You *cannot* use the C++ scope resolution operator '::', nor the C/C++ member operators '.' or '->' in embedded SQL statements. You can easily accomplish the same thing through use of local pointer or reference variables, which are set outside the SQL statement, to point to the desired scoped variable, then used inside the SQL statement to refer to it. The following example shows the correct method to use:

```
EXEC SQL BEGIN DECLARE SECTION;
  char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
  SELECT name INTO :localName FROM STAFF
  WHERE name = 'Sanders';
```

Multi-Byte Character Encoding in C and C++

Some character encoding schemes, particularly those from east Asian countries, require multiple bytes to represent a character. This external representation of data is called the *multi-byte character code* representation of a character, and includes double-byte characters (characters represented by two bytes). Graphic data in DB2® consists of double-byte characters.

To manipulate character strings with double-byte characters, it may be convenient for an application to use an internal representation of data. This internal representation is called the *wide-character code* representation of the double-byte

characters, and is the format customarily used in the wchar_t C/C++ data type. Subroutines that conform to ANSI C and X/OPEN Portability Guide 4 (XPG4) are available to process wide-character data, and to convert data in wide-character format to and from multibyte format.

Note that although an application can process character data in either multibyte format or wide-character format, interaction with the database manager is done with DBCS (multibyte) character codes only. That is, data is stored in and retrieved from GRAPHIC columns in DBCS format. The WCHARTYPE precompiler option is provided to allow application data in wide-character format to be converted to/from multibyte format when it is exchanged with the database engine.

Related concepts:

- "Graphic Host Variables in C and C++" on page 143
- "wchar_t and sqldbchar Data Types in C and C++" on page 157

wchar_t and sqldbchar Data Types in C and C++

While the size and encoding of DB2® graphic data is constant from one platform to another for a particular code page, the size and internal format of the ANSI C or C++ wchar_t data type depends on which compiler you use and which platform you are on. The sqldbchar data type, however, is defined by DB2 to be two bytes in size, and is intended to be a portable way of manipulating DBCS and UCS-2 data in the same format in which it is stored in the database.

You can define all DB2 C graphic host variable types using either wchar_t or sqldbchar. You must use wchar_t if you build your application using the WCHARTYPE CONVERT precompile option.

Note: When specifying the WCHARTYPE CONVERT option on a Windows® platform, you should note that wchar_t on Windows platforms is Unicode. Therefore, if your C/C++ compiler's wchar_t is not Unicode, the wcstombs() function call may fail with SQLCODE -1421 (SQLSTATE=22504). If this happens, you can specify the WCHARTYPE NOCONVERT option, and explicitly call the wcstombs() and mbstowcs() functions from within your program.

If you build your application with the WCHARTYPE NOCONVERT precompile option, you should use sqldbchar for maximum portability between different DB2 client and server platforms. You may use wchar_t with WCHARTYPE NOCONVERT, but only on platforms where wchar_t is defined as two bytes in length.

If you incorrectly use either wchar_t or sqldbchar in host variable declarations, you will receive an SQLCODE 15 (no SQLSTATE) at precompile time.

Related concepts:

- "WCHARTYPE Precompiler Option in C and C++" on page 158
- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 614

WCHARTYPE Precompiler Option in C and C++

Using the WCHARTYPE precompiler option, you can specify which graphic character format you want to use in your C/C++ application. This option provides you with the flexibility to choose between having your graphic data in multibyte format or in wide-character format. There are two possible values for the WCHARTYPE option:

CONVERT

If you select the WCHARTYPE CONVERT option, character codes are converted between the graphic host variable and the database manager. For graphic input host variables, the character code conversion from wide-character format to multibyte DBCS character format is performed before the data is sent to the database manager, using the ANSI C function `wcstombs()`. For graphic output host variables, the character code conversion from multibyte DBCS character format to wide-character format is performed before the data received from the database manager is stored in the host variable, using the ANSI C function `mbstowcs()`.

The advantage to using WCHARTYPE CONVERT is that it allows your application to fully exploit the ANSI C mechanisms for dealing with wide-character strings (L-literals, 'wc' string functions, and so on) without having to explicitly convert the data to multibyte format before communicating with the database manager. The disadvantage is that the implicit conversions may have an impact on the performance of your application at run time, and may increase memory requirements.

If you select WCHARTYPE CONVERT, declare all graphic host variables using `wchar_t` instead of `sqldbchar`.

If you want WCHARTYPE CONVERT behavior, but your application does not need to be precompiled (for example, a CLI application), then define the C preprocessor macro `SQL_WCHART_CONVERT` at compile time. This ensures that certain definitions in the DB2 header files use the data type `wchar_t` instead of `sqldbchar`.

Note: The WCHARTYPE CONVERT precompile option is not currently supported in programs running on the DB2® Windows® 3.1 client. For those programs, use the default (WCHARTYPE NOCONVERT).

NOCONVERT (default)

If you choose the WCHARTYPE NOCONVERT option, or do not specify any WCHARTYPE option, no implicit character code conversion occurs between the application and the database manager. Data in a graphic host variable is sent to and received from the database manager as unaltered DBCS characters. This has the advantage of improved performance, but the disadvantage that your application must either refrain from using wide-character data in `wchar_t` host variables, or must explicitly call the `wcstombs()` and `mbstowcs()` functions to convert the data to and from multibyte format when interfacing with the database manager.

If you select WCHARTYPE NOCONVERT, declare all graphic host variables using the `sqldbchar` type for maximum portability to other DB2 client/server platforms.

Other guidelines you need to observe are:

- Because `wchar_t` or `sqldbchar` support is used to handle DBCS data, its use requires DBCS or EUC capable hardware and software. This support is only

available in the DBCS environment of DB2 Universal Database, or for dealing with GRAPHIC data in any application (including single-byte applications) connected to a UCS-2 database.

- Non-DBCS characters, and wide-characters that can be converted to non-DBCS characters, should not be used in graphic strings. *Non-DBCS characters* refers to single-byte characters, and non-double byte characters. Graphic strings are not validated to ensure that their values contain only double-byte character code points. Graphic host variables must contain only DBCS data, or, if WCHARTYPE CONVERT is in effect, wide-character data that converts to DBCS data. You should store mixed double-byte and single-byte data in character host variables. Note that mixed data host variables are unaffected by the setting of the WCHARTYPE option.
- In applications where the WCHARTYPE NOCONVERT precompile option is used, L-literals should not be used in conjunction with graphic host variables, because L-literals are in wide-character format. An L-literal is a C wide-character string literal prefixed by the letter L which has the data type `"array of wchar_t"`. For example, `L"dbcs-string"` is an L-literal.
- In applications where the WCHARTYPE CONVERT precompile option is used, L-literals can be used to initialize `wchar_t` host variables, but cannot be used in SQL statements. Instead of using L-literals, SQL statements should use graphic string constants, which are independent of the WCHARTYPE setting.
- The setting of the WCHARTYPE option affects graphic data passed to and from the database manager using the SQLDA structure as well as host variables. If WCHARTYPE CONVERT is in effect, graphic data received from the application through an SQLDA will be presumed to be in wide-character format, and will be converted to DBCS format via an implicit call to `wcstombs()`. Similarly, graphic output data received by an application will have been converted to wide-character format before being placed in application storage.
- Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. Ordinary fenced stored procedures may be precompiled with either the CONVERT or NOCONVERT options, which will affect the format of graphic data manipulated by SQL statements contained in the stored procedure. In either case, however, any graphic data passed into the stored procedure through the SQLDA will be in DBCS format. Likewise, data passed out of the stored procedure through the SQLDA must be in DBCS format.
- If an application calls a stored procedure through the Database Application Remote Interface (DARI) interface (the `sqleproc()` API), any graphic data in the input SQLDA must be in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the state of the calling application's WCHARTYPE setting. Likewise, any graphic data in the output SQLDA will be returned in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the WCHARTYPE setting.
- If an application calls a stored procedure through the SQL CALL statement, graphic data conversion will occur on the SQLDA, depending on the calling application's WCHARTYPE setting.
- Graphic data passed to user-defined functions (UDFs) will always be in DBCS format. Likewise, any graphic data returned from a UDF will be assumed to be in DBCS format for DBCS databases, and UCS-2 format for EUC and UCS-2 databases.
- Data stored in DBCLOB files through the use of DBCLOB file reference variables is stored in either DBCS format, or, in the case of UCS-2 databases, in UCS-2 format. Likewise, input data from DBCLOB files is retrieved either in DBCS format, or, in the case of UCS-2 databases, in UCS-2 format.

Note: If you precompile C applications using the WCHARTYPE CONVERT option, DB2 validates the applications' graphic data on both input and output as the data is passed through the conversion functions. If you do *not* use the CONVERT option, no conversion of graphic data, and hence no validation occurs. In a mixed CONVERT/NOCONVERT environment, this may cause problems if invalid graphic data is inserted by a NOCONVERT application and then fetched by a CONVERT application. This data fails the conversion with an SQLCODE -1421 (SQLSTATE 22504) on a FETCH in the CONVERT application.

Related reference:

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

Japanese or Traditional Chinese EUC, and UCS-2 Considerations in C and C++

If your application code page is Japanese or Traditional Chinese EUC, or if your application connects to a UCS-2 database, you can access GRAPHIC columns at a database server by using either the CONVERT or the NOCONVERT option and `wchar_t` or `sqldbchar` graphic host variables, or input/output SQLDAs. In this section, *DBCS format* refers to the UCS-2 encoding scheme for EUC data. Consider the following cases:

- CONVERT option used

  The DB2® client converts graphic data from the wide character format to your application code page, then to UCS-2 before sending the input SQLDA to the database server. Any graphic data is sent to the database server tagged with the UCS-2 code page identifier. Mixed character data is tagged with the application code page identifier. When graphic data is retrieved from a database by a client, it is tagged with the UCS-2 code page identifier. The DB2 client converts the data from UCS-2 to the client application code page, then to the wide character format. If an input SQLDA is used instead of a host variable, you are required to ensure that graphic data is encoded using the wide character format. This data will be converted to UCS-2, then sent to the database server. These conversions will impact performance.
- NOCONVERT option used

  The graphic data is assumed by DB2 to be encoded using UCS-2 and is tagged with the UCS-2 code page, and no conversions are done. DB2 assumes that the graphic host variable is being used simply as a bucket. When the NOCONVERT option is chosen, graphic data retrieved from the database server is passed to the application encoded using UCS-2. Any conversions from the application code page to UCS-2 and from UCS-2 to the application code page are your responsibility. Data tagged as UCS-2 is sent to the database server without any conversions or alterations.

To minimize conversions you can either use the NOCONVERT option and handle the conversions in your application, or not use GRAPHIC columns. For the client environments where `wchar_t` encoding is in two-byte Unicode, for example Windows® NT or AIX® version 4.3 and higher, you can use the NOCONVERT option and work directly with UCS-2. In such cases, your application should handle the difference between big-endian and little-endian architectures. With the NOCONVERT option, DB2 Universal Database uses `sqldbchar`, which is always two-byte big-endian.

Do not assign IBM®-eucJP/IBM-eucTW CS0 (7-bit ASCII) and IBM-eucJP CS2 (Katakana) data to graphic host variables either after conversion to UCS-2 (if NOCONVERT is specified) or by conversion to the wide character format (if CONVERT is specified). The reason is that characters in both of these EUC code sets become single-byte when converted from UCS-2 to PC DBCS.

In general, although eucJP and eucTW store GRAPHIC data as UCS-2, the GRAPHIC data in these databases is still non-ASCII eucJP or eucTW data. Specifically, any space padded to such GRAPHIC data is DBCS space (also known as ideographic space in UCS-2, U+3000). For a UCS-2 database, however, GRAPHIC data can contain any UCS-2 character, and space padding is done with UCS-2 space, U+0020. Keep this difference in mind when you code applications to retrieve UCS-2 data from a UCS-2 database versus UCS-2 data from eucJP and eucTW databases.

Related concepts:

- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 614

SQL Declare Section with Host Variables for C and C++

The following is a sample SQL declare section with host variables declared for supported SQL data types:

```
EXEC SQL BEGIN DECLARE SECTION;

:
        short     age = 26;               /* SQL type  500 */
        short     year;                   /* SQL type  500 */
        sqlint32  salary;                 /* SQL type  496 */
        sqlint32  deptno;                 /* SQL type  496 */
        float     bonus;                  /* SQL type  480 */
        double    wage;                   /* SQL type  480 */
        char      mi;                     /* SQL type  452 */
        char      name[6];                /* SQL type  460 */
        struct   {
                  short len;
                  char data[24];
                 } address;               /* SQL type  448 */
        struct   {
                  short len;
                  char data[32695];
                 } voice;                 /* SQL type  456 */
        sql type is clob(1m)
                  chapter;                /* SQL type  408 */
        sql type is clob_locator
                  chapter_locator;        /* SQL type  964 */
        sql type is clob_file
                  chapter_file_ref;       /* SQL type  920 */
        sql type is blob(1m)
                  video;                  /* SQL type  404 */
        sql type is blob_locator
                  video_locator;          /* SQL type  960 */
        sql type is blob_file
                  video_file_ref;         /* SQL type  916 */
        sql type is dbclob(1m)
                  tokyo_phone_dir;        /* SQL type  412 */
        sql type is dbclob_locator
                  tokyo_phone_dir_lctr;   /* SQL type  968 */
        sql type is dbclob_file
                  tokyo_phone_dir_flref;  /* SQL type  924 */
```

```
        struct   {
                  short len;
                  sqldbchar data[100];
                 } vargraphic1;           /* SQL type  464 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        struct   {
                  short len;
                  wchar_t data[100];
                 } vargraphic2;           /* SQL type  464 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        struct   {
                  short len;
                  sqldbchar data[10000];
                 } long_vargraphic1;      /* SQL type  472 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        struct   {
                  short len;
                  wchar_t data[10000];
                 } long_vargraphic2;      /* SQL type  472 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        sqldbchar graphic1[100];          /* SQL type  468 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        wchar_t   graphic2[100];          /* SQL type  468 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        char      date[11];               /* SQL type  384 */
        char      time[9];                /* SQL type  388 */
        char      timestamp[27];          /* SQL type  392 */
        short     wage_ind;               /* Null indicator */

.
.
.
   EXEC SQL END DECLARE SECTION;
```

Data Type Considerations for C and C++

The sections that follow describe how SQL data types map to C and C++ data types.

Supported SQL Data Types in C and C++

Certain predefined C and C++ data types correspond to the database manager column types. Only these C/C++ data types can be declared as host variables.

The following table shows the C/C++ equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

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

Table 13. SQL Data Types Mapped to C/C++ Declarations

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short short int sqlint16	16-bit signed integer
INTEGER (496 or 497)	long long int sqlint32[2]	32-bit signed integer
BIGINT (492 or 493)	long long long __int64 sqlint64[3]	64-bit signed integer
REAL[4] (480 or 481)	float	Single-precision floating point
DOUBLE[5] (480 or 481)	double	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	No exact equivalent; use double	Packed decimal (Consider using the CHAR and DECIMAL functions to manipulate packed decimal fields as character data.)
CHAR(1) (452 or 453)	char	Single character
CHAR(*n*) (452 or 453)	No exact equivalent; use char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length character string

VARCHAR(*n*)
(448 or 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		Non null-terminated varying character string with 2-byte string length indicator

	Alternatively, use char[*n+1*] where n is large enough to hold the data 1<=*n*<=32 672	Null-terminated variable-length character string **Note:** Assigned an SQL type of 460/461.

LONG VARCHAR
(456 or 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		Non null-terminated varying character string with 2-byte string length indicator

CLOB(*n*) (408 or 409)	sql type is clob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying character string with 4-byte string length indicator
CLOB locator variable[6] (964 or 965)	sql type is clob_locator	Identifies CLOB entities residing on the server
CLOB file reference variable[6] (920 or 921)	sql type is clob_file	Descriptor for file containing CLOB data

Table 13. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description
BLOB(*n*) (404 or 405)	sql type is blob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying binary string with 4-byte string length indicator
BLOB locator variable[6] (960 or 961)	sql type is blob_locator	Identifies BLOB entities on the server
BLOB file reference variable[6] (916 or 917)	sql type is blob_file	Descriptor for the file containing BLOB data
DATE (384 or 385)	Null-terminated character form	Allow at least 11 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 10 characters.
TIME (388 or 389)	Null-terminated character form	Allow at least 9 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 8 characters.
TIMESTAMP (392 or 393)	Null-terminated character form	Allow at least 27 characters to accommodate the null-terminator.
	VARCHAR structured form	Allow at least 26 characters.
Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.		
GRAPHIC(1) (468 or 469)	sqldbchar	Single double-byte character
GRAPHIC(*n*) (468 or 469)	No exact equivalent; use sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length double-byte character string

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 336
		Non null-terminated varying double-byte character string with 2-byte string length indicator

	Alternatively use sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336	Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC
(472 or 473)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		Non null-terminated varying double-byte character string with 2-byte string length indicator

Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE CONVERT option.		
GRAPHIC(1) (468 or 469)	wchar_t	• Single wide character (for C-type) • Single double-byte character (for column type)

Table 13. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description
GRAPHIC(*n*) (468 or 469)	No exact equivalent; use wchar_t [*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length double-byte character string

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	1<=*n*<=16 336
		Non null-terminated varying double-byte character string with 2-byte string length indicator

	Alternately use char[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336	Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC
(472 or 473)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	16 337<=*n*<=16 350
		Non null-terminated varying double-byte character string with 2-byte string length indicator

Note: The following data types are only available in the DBCS or EUC environment.		
DBCLOB(*n*) (412 or 413)	sql type is dbclob(*n*) 1<=*n*<=1 073 741 823	Non null-terminated varying double-byte character string with 4-byte string length indicator
DBCLOB locator variable[6] (968 or 969)	sql type is dbclob_locator	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[6] (924 or 925)	sql type is dbclob_file	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. For platform compatibility, use sqlint32. On 64-bit UNIX platforms, "long" is a 64 bit integer. On 64-bit Windows operating systems and 32-bit UNIX platforms "long" is a 32 bit integer.
3. For platform compatibility, use sqlint64. The DB2 Universal Database sqlsystm.h header file will type define sqlint64 as "__int64" on the Windows NT platform when using the Microsoft compiler, "long long" on 32-bit UNIX platforms, and "long" on 64 bit UNIX platforms.
4. FLOAT(*n*) where 0 < *n* < 25 is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where 24 < *n* < 54 is
   - DOUBLE PRECISION
6. This is not a column type but a host variable type.

The following are additional rules for supported C/C++ data types:

- The data type `char` can be declared as `char` or `unsigned char`.
- The database manager processes null-terminated variable-length character string data type char[*n*] (data type 460), as VARCHAR(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in char[*n*] or the number of bytes preceding the first null-terminator (\0), whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of bytes preceding the first null-terminator (\0).
- The database manager processes null-terminated, variable-length graphic string data type, `wchar_t[n]` or `sqldbchar[n]` (data type 400), as VARGRAPHIC(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in `wchar_t[n]` or `sqldbchar[n]`, or the number of characters preceding the first graphic null-terminator, whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of characters preceding the first graphic null-terminator.
- Unsigned numeric data types are not supported.
- The C/C++ data type `int` is not allowed because its internal representation is machine dependent.

Related concepts:

- "SQL Declare Section with Host Variables for C and C++" on page 161

FOR BIT DATA in C and C++

The standard C or C++ string type 460 should not be used for columns designated FOR BIT DATA. The database manager truncates this data type when a null character is encountered. Use either the VARCHAR (SQL type 448) or CLOB (SQL type 408) structures.

Related concepts:

- "SQL Declare Section with Host Variables for C and C++" on page 161

Related reference:

- "Supported SQL Data Types in C and C++" on page 162

C and C++ Data Types for Procedures, Functions, and Methods

The following table lists the supported mappings between SQL data types and C data types for procedures, UDFs, and methods.

Table 14. SQL Data Types Mapped to C/C++ Declarations

SQL Column Type	C/C++ Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short	16-bit signed integer
INTEGER (496 or 497)	sqlint32	32-bit signed integer
BIGINT (492 or 493)	sqlint64	64-bit signed integer
REAL (480 or 481)	float	Single-precision floating point

Table 14. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type	C/C++ Data Type	SQL Column Type Description
DOUBLE (480 or 481)	double	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	Not supported.	To pass a decimal value, define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type.
CHAR(*n*) (452 or 453)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length, null-terminated character string
CHAR(*n*) FOR BIT DATA (452 or 453)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length character string
VARCHAR(*n*) (448 or 449) (460 or 461)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=32 672	Null-terminated varying length string

VARCHAR(*n*) FOR BIT DATA
(448 or 449)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
		Not null-terminated varying length character string

LONG VARCHAR
(456 or 457)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		Not null-terminated varying length character string

CLOB(*n*)
(408 or 409)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		Not null-terminated varying length character string with 4-byte string length indicator

BLOB(*n*)
(404 or 405)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		Not null-terminated varying binary string with 4-byte string length indicator

DATE (384 or 385)	char[11]	Null-terminated character form
TIME (388 or 389)	char[9]	Null-terminated character form
TIMESTAMP (392 or 393)	char[27]	Null-terminated character form
Note: The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.		
GRAPHIC(*n*) (468 or 469)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=127	Fixed-length, null-terminated double-byte character string
VARGRAPHIC(*n*) (400 or 401)	sqldbchar[*n+1*] where n is large enough to hold the data 1<=*n*<=16 336	Not null-terminated, variable-length double-byte character string

Table 14. SQL Data Types Mapped to C/C++ Declarations (continued)

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

LONG VARGRAPHIC
(472 or 473)
	struct {
	sqluint16 length;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		Not null-terminated, variable-length double-byte character string

DBCLOB(*n*)
(412 or 413)
	struct {
	sqluint32 length;
	sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
		Not null-terminated varying length character string with 4-byte string length indicator

SQLSTATE and SQLCODE Variables in C and C++

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION;
  char      SQLSTATE[6]
  sqlint32  SQLCODE;


⋮
  EXEC SQL END DECLARE SECTION;
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

In an application that is made up of multiple source files, the SQLCODE and SQLSTATE variables may be defined in the first source file as above. Subsequent source files should modify the definitions as follows:

```
extern sqlint32 SQLCODE;
extern char     SQLSTATE[6];
```

Related concepts:

- "Return Codes" on page 99
- "Error Information in the SQLCODE, SQLSTATE, and SQLWARN Fields" on page 100

Chapter 7. Multiple-Thread Database Access for C and C++ Applications

Purpose of Multiple-Thread Database Access

One feature of some operating systems is the ability to run several threads of execution within a single process. The multiple threads allow an application to handle asynchronous events, and makes it easier to create event-driven applications, without resorting to polling schemes. The information that follows describes how the database manager works with multiple threads, and lists some design guidelines that you should keep in mind.

If you are not familiar with terms relating to the development of multithreaded applications (such as critical section and semaphore), consult the programming documentation for your operating system.

A DB2 application can execute SQL statements from multiple threads using *contexts*. A context is the environment from which an application runs all SQL statements and API calls. All connections, units of work, and other database resources are associated with a specific context. Each context is associated with one or more threads within an application.

For each executable SQL statement in a context, the first run-time services call always tries to obtain a latch. If it is successful, it continues processing. If not (because an SQL statement in another thread of the same context already has the latch), the call is blocked on a signaling semaphore until that semaphore is posted, at which point the call gets the latch and continues processing. The latch is held until the SQL statement has completed processing, at which time it is released by the last run-time services call that was generated for that particular SQL statement.

The net result is that each SQL statement within a context is executed as an atomic unit, even though other threads may also be trying to execute SQL statements at the same time. This action ensures that internal data structures are not altered by different threads at the same time. APIs also use the latch used by run-time services; therefore, APIs have the same restrictions as run-time services routines within each context.

For DB2® Version 8, all Version 8 applications are multithreaded by default, and are capable of using multiple contexts. (The behavior of pre-Version 8 applications remains unchanged.) If you want, you can use the following DB2 APIs to use multiple contexts. Specifically, your application can create a context for a thread, attach to or detach from a separate context for each thread, and pass contexts between threads. If your application does not call *any* of these APIs, DB2 will automatically manage the multiple contexts for your application:

- `sqleBeginCtx()`
- `sqleEndCtx()`
- `sqleAttachToCtx()`

- `sqleDetachFromCtx()`
- `sqleGetCurrentCtx()`
- `sqleInterruptCtx()`

Contexts may be exchanged between threads in a process, but not exchanged between processes. One use of multiple contexts is to provide support for concurrent transactions.

Related concepts:

- "Concurrent Transactions" on page 633

Related reference:

- "sqleAttachToCtx - Attach to Context" in the *Administrative API Reference*
- "sqleBeginCtx - Create and Attach to an Application Context" in the *Administrative API Reference*
- "sqleDetachFromCtx - Detach From Context" in the *Administrative API Reference*
- "sqleEndCtx - Detach and Destroy Application Context" in the *Administrative API Reference*
- "sqleGetCurrentCtx - Get Current Context" in the *Administrative API Reference*
- "sqleInterruptCtx - Interrupt Context" in the *Administrative API Reference*

Related samples:

- "dbthrds.sqc -- How to use multiple context APIs on UNIX (C)"
- "dbthrds.sqC -- How to use multiple context APIs on UNIX (C++)"

Recommendations for Using Multiple Threads

Follow these guidelines when accessing a database from multiple thread applications:

- Serialize alteration of data structures.

  Applications must ensure that user-defined data structures used by SQL statements and database manager routines are not altered by one thread while an SQL statement or database manager routine is being processed in another thread. For example, do not allow a thread to reallocate an SQLDA while it was being used by an SQL statement in another thread.

- Consider using separate data structures.

  It may be easier to give each thread its own user-defined data structures to avoid having to serialize their usage. This guideline is especially true for the SQLCA, which is used not only by every executable SQL statement, but also by all of the database manager routines. There are three alternatives for avoiding this problem with the SQLCA:

  – Use EXEC SQL INCLUDE SQLCA, but add `struct sqlca sqlca` at the beginning of any routine that is used by any thread other than the first thread.
  – Place EXEC SQL INCLUDE SQLCA inside each routine that contains SQL, instead of in the global scope.
  – Replace EXEC SQL INCLUDE SQLCA with `#include "sqlca.h"`, then add `"struct sqlca sqlca"` at the beginning of any routine that uses SQL.

Note: It is recommended that you do not use the default stack size, but instead increase the stack size to at least 256 000. DB2® requires a minimum stack

size of 256 000 when calling a DB2 function. You must ensure therefore, that you allocate a total stack size that is large enough for both your application and the minimum requirements for a DB2 function call.

Code Page and Country/Region Code Considerations for Multithreaded UNIX Applications

On AIX®, the Solaris Operating Environment, HP-UX, and Silicon Graphics IRIX, changes have been made to the functions that are used for run-time querying of the code page and country/region code to be used for a database connection. These functions are now thread safe, but can create some lock contention (and resulting performance degradation) in a multithreaded application that uses a large number of concurrent database connections.

You can use the DB2®_FORCE_NLS_CACHE environment variable to eliminate the chance of lock contention in multithreaded applications. When DB2_FORCE_NLS_CACHE is set to TRUE, the code page and country/region code information is saved the first time a thread accesses it. From that point on, the cached information will be used for any other thread that requests this information. By saving this information, lock contention is eliminated, and in certain situations a performance benefit will be realized.

You should not set DB2_FORCE_NLS_CACHE to TRUE if the application changes locale settings between connections. If this situation occurs, the original locale information will be returned even after the locale settings have been changed. In general, multithreaded applications will not change locale settings, which, ensures that the application remains thread safe.

Related concepts:

- "DB2 registry and environment variables" in the *Administration Guide: Performance*

Troubleshooting Multithreaded Applications

The sections that follow describe problems that can occur with multithreaded application, and how to avoid them.

Potential Problems with Multiple Threads

An application that uses multiple threads is, understandably, more complex than a single-threaded application. This extra complexity can potentially lead to some unexpected problems. When writing a multithreaded application, exercise caution with the following:

- Database dependencies between two or more contexts.

  Each context in an application has its own set of database resources, including locks on database objects. This characteristic makes it possible for two contexts, if they are accessing the same database object, to deadlock. The database manager will detect the deadlock. One of the contexts will receive SQLCODE -911 and its unit of work will be rolled back.
- Application dependencies between two or more contexts.

  Be careful with any programming techniques that establish inter-context dependencies. Latches, semaphores, and critical sections are examples of programming techniques that can establish such dependencies. If an application

has two contexts that have both application and database dependencies between the contexts, it is possible for the application to become deadlocked. If some of the dependencies are outside of the database manager, the deadlock is not detected, thus the application gets suspended or hung.

Related concepts:

- "Deadlock Prevention for Multiple Contexts" on page 172

Deadlock Prevention for Multiple Contexts

Because the database manager cannot detect deadlocks between threads, design and code your application in a way that will prevent (or at least avoid) deadlocks.

As an example of a deadlock that the database manager would not detect, consider an application that has two contexts, both of which access a common data structure. To avoid problems where both contexts change the data structure simultaneously, the data structure is protected by a semaphore. The contexts look like this:

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT

context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```

Suppose the first context successfully executes the SELECT and the UPDATE statements, while the second context gets the semaphore and accesses the data structure. The first context now tries to get the semaphore, but it cannot because the second context is holding the semaphore. The second context now attempts to read a row from table TAB1, but it stops on a database lock held by the first context. The application is now in a state where context 1 cannot finish before context 2 is done and context 2 is waiting for context 1 to finish. The application is deadlocked, but because the database manager does not know about the semaphore dependency neither context will be rolled back. The unresolved dependency leaves the application suspended.

You can avoid the deadlock that would occur for the previous example in several ways.

- Release all locks held before obtaining the semaphore.

  Change the code for context 1 to perform a commit before it gets the semaphore.
- Do not code SQL statements inside a section protected by semaphores.

  Change the code for context 2 to release the semaphore before doing the SELECT.
- Code all SQL statements within semaphores.

  Change the code for context 1 to obtain the semaphore before running the SELECT statement. While this technique will work, it is not highly recommended because the semaphores will serialize access to the database manager, which potentially negates the benefits of using multiple threads.

- Set the *locktimeout* database configuration parameter to a value other than -1.

  While a value other than -1 will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. When handling the roll back error, context 2 should release the semaphore. Once the semaphore has been released, context 1 can continue and context 2 is free to retry its work.

The techniques for avoiding deadlocks are described in terms of the example, but you can apply them to all multithreaded applications. In general, treat the database manager as you would treat any protected resource and you should not run into problems with multithreaded applications.

Related concepts:

- "Potential Problems with Multiple Threads" on page 171

Chapter 8. Programming in COBOL

Programming Considerations for COBOL

Special host-language programming considerations are discussed in the following sections. Included is information on language restrictions, host language specific include files, embedding SQL statements, host variables, and supported data types for host variables. See the Micro Focus COBOL documentation for information about embedding SQL statements, language restrictions, and supported data types for host variables.

Related reference:

- "COBOL samples" in the *Application Development Guide: Building and Running Applications*

Language Restrictions in COBOL

All API pointers are 4 bytes long. All integer variables used as value parameters in API calls must be declared with a USAGE COMP-5 clause.

Multiple-Thread Database Access in COBOL

COBOL does not support multiple-thread database access.

Input and Output Files for COBOL

By default, the input file has an extension of `.sqb`, but if you use the TARGET precompile option (TARGET ANSI_COBOL, TARGET IBMCOB, TARGET MFCOB or TARGET MFCOB16), the input file can have any extension you prefer.

By default, the output file has an extension of `.cbl`, but you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

Include Files for COBOL

The host-language-specific include files for COBOL have the file extension `.cbl`. If you use the ″System/390 host data type support″ feature of IBM COBOL compiler, the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_i
```

If you build the DB2 sample programs with the supplied script files, you must change the include file path specified in the script files to the `cobol_i` directory and not the `cobol_a` directory.

If you do **not** use the ″System/390 host data type support″ feature of the IBM COBOL compiler, or you use an earlier version of this compiler, the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_a
```

The include files that are intended to be used in your applications are described below.

SQL (sql.cbl) This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLAPREP (sqlaprep.cbl)
This file contains definitions required to write your own precompiler.

SQLCA (sqlca.cbl)
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

SQLCA_92 (sqlca_92.cbl)
This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of the `sqlca.cbl` file when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.cbl` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

SQLCODES (sqlcodes.cbl)
This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqlda.cbl)
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

SQLEAU (sqleau.cbl)
This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

SQLENV (sqlenv.cbl)
This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLETSD (sqletsd.cbl)
This file defines the Table Space Descriptor structure, SQLETSDESC, which is passed to the Create Database API, sqlgcrea.

SQLE819A (sqle819a.cbl)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.cbl)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.cbl)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.cbl)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.cbl)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.cbl)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQL1252A (sql1252a.cbl)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQL1252B (sql1252b.cbl)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLMON (sqlmon.cbl)
This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLMONCT (sqlmonct.cbl)
This file contains constant definitions and local data structure definitions required to call the Database System Monitor APIs.

SQLSTATE (sqlstate.cbl)
This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLUTBCQ (sqlutbcq.cbl)
This file defines the Table Space Container Query data structure, SQLB-TBSCONTQRY-DATA, which is used with the table space container query APIs, sqlgstsc, sqlgftcq, and sqlgtcq.

SQLUTBSQ (sqlutbsq.cbl)
This file defines the Table Space Query data structure, SQLB-TBSQRY-DATA, which is used with the table space query APIs, sqlgstsq, sqlgftsq, and sqlgtsq.

SQLUTIL (sqlutil.cbl)
This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

Embedded SQL Statements in COBOL

Embedded SQL statements consist of the following three elements:

Element	Correct COBOL Syntax
Keyword pair	EXEC SQL
Statement string	Any valid SQL statement
Statement terminator	END-EXEC.

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

The following rules apply to embedded SQL statements:

- Executable SQL statements must be placed in the PROCEDURE DIVISION. The SQL statements can be preceded by a paragraph name, just as a COBOL statement.
- SQL statements can begin in either Area A (columns 8 through 11) or Area B (columns 12 through 72).
- Start each SQL statement with EXEC SQL and end it with END-EXEC. The SQL precompiler includes each SQL statement as a comment in the modified source file.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (`--`), followed by a string of zero or more characters and terminated by a line end. Do not place SQL

comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the COBOL language.

- COBOL comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
- SQL statements follow the same line continuation rules as the COBOL language. However, do not split the EXEC SQL keyword pair between lines.
- Do not use the COBOL COPY statement to include files containing SQL statements. SQL statements are precompiled before the module is compiled. The precompiler will ignore the COBOL COPY statement. Instead, use the SQL INCLUDE statement to include these files.

  To locate the INCLUDE file, the DB2® COBOL precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

  - `EXEC SQL INCLUDE payroll END-EXEC.`

    If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for `payroll.sqb`, then `payroll.cpy`, then `payroll.cbl`, in each directory in which it looks.
  - `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

    If the file name is enclosed in quotation marks, as above, no extension is added to the name.

    If the file name in quotation marks does not contain an absolute path, the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for AIX, if DB2INCLUDE is set to '`/disk2:myfiles/cobol`', the precompiler searches for '`./pay/payroll.cbl`', then '`/disk2/pay/payroll.cbl`', and finally '`./myfiles/cobol/pay/payroll.cbl`'. The path where the file is actually found is displayed in the precompiler messages. On Windows® platforms, substitute back slashes (\) for the forward slashes in the above example.

  Note: The setting of DB2INCLUDE is cached by the DB2 command line processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.
- To continue a string constant to the next line, column 7 of the continuing line must contain a '-' and column 12 or beyond must contain a string delimiter.
- SQL arithmetic operators must be delimited by blanks.
- Full-line COBOL comments can occur anywhere in the program, including within SQL statements.
- Use host variables exactly as declared when referencing host variables in an SQL statement.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a COBOL program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, Windows-based platforms use Carriage Return/Line Feed for end-of-line, whereas UNIX®-based systems use just a Line Feed.

Related reference:

- Appendix A, "Supported SQL Statements," on page 685

Host Variables in COBOL

The sections that follow describe how to declare and use host variables in COBOL programs.

Host Variables in COBOL

Host variables are COBOL language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other COBOL variable.

Related concepts:

- "Host Variable Names in COBOL" on page 180
- "Host Variable Declarations in COBOL" on page 181

Related reference:

- "Syntax for Numeric Host Variables in COBOL" on page 181
- "Syntax for Fixed-Length Character Host Variables in COBOL" on page 182
- "Syntax for Fixed-Length Graphic Host Variables in COBOL" on page 183
- "Syntax for LOB Host Variables in COBOL" on page 184
- "Syntax for LOB Locator Host Variables in COBOL" on page 185
- "Syntax for File Reference Host Variables in COBOL" on page 186

Host Variable Names in COBOL

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than `SQL`, `sql`, `DB2`®, or `db2`, which are reserved for system use.
- FILLER items using the declaration syntaxes described below are permitted in group host variable declarations, and will be ignored by the precompiler. However, if you use FILLER more than once within an SQL DECLARE section, the precompiler fails. You may not include FILLER items in VARCHAR, LONG VARCHAR, VARGRAPHIC or LONG VARGRAPHIC declarations.
- You can use hyphens in host variable names.

  SQL interprets a hyphen enclosed by spaces as a subtraction operator. Use hyphens without spaces in host variable names.
- The REDEFINES clause is permitted in host variable declarations.
- Level-88 declarations are permitted in the host variable declare section, but are ignored.

Related concepts:

- "Host Variable Declarations in COBOL" on page 181

Related reference:

- "Syntax for Numeric Host Variables in COBOL" on page 181

Host Variable Declarations in COBOL

An SQL declare section must be used to identify host variable declarations. This section alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The COBOL precompiler only recognizes a subset of valid COBOL declarations.

Related tasks:

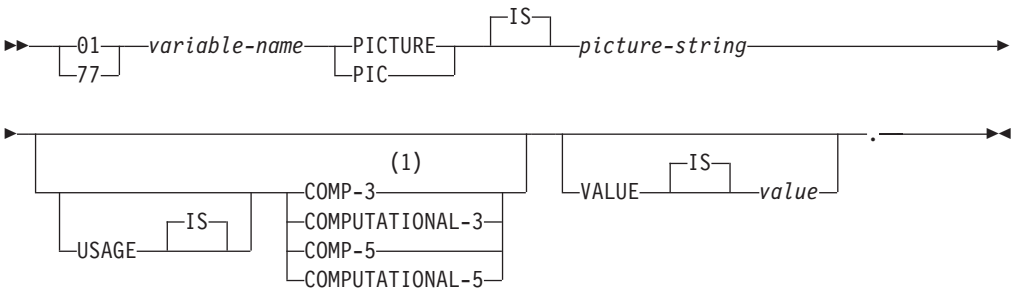
- "Declaring structured type host variables" in the *Application Development Guide: Programming Server Applications*

Related reference:

Syntax for Numeric Host Variables in COBOL

Following is the syntax for numeric host variables.

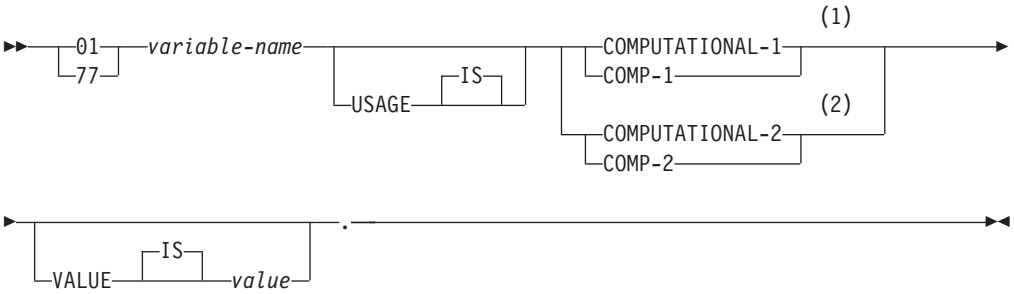
Syntax for Numeric Host Variables in COBOL



Notes:

1 An alternative for COMP-3 is PACKED-DECIMAL.

Floating Point



Notes:

1 REAL (SQLTYPE 480), Length 4

2 DOUBLE (SQLTYPE 480), Length 8

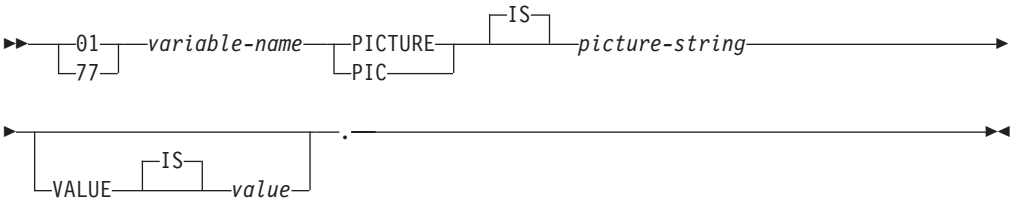
Numeric Host Variable Considerations:

1. *Picture-string* must have one of the following forms:
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. Nines may be expanded (for example., ″S999″ instead of S9(3)″)
3. *m* and *n* must be positive integers.

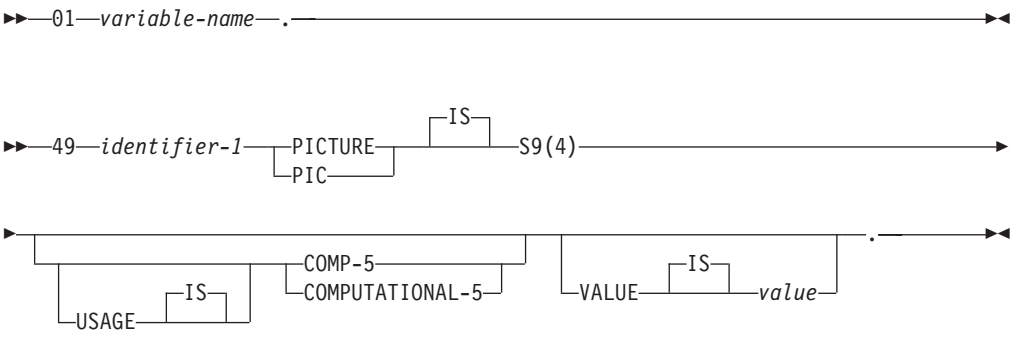
Syntax for Fixed-Length Character Host Variables in COBOL

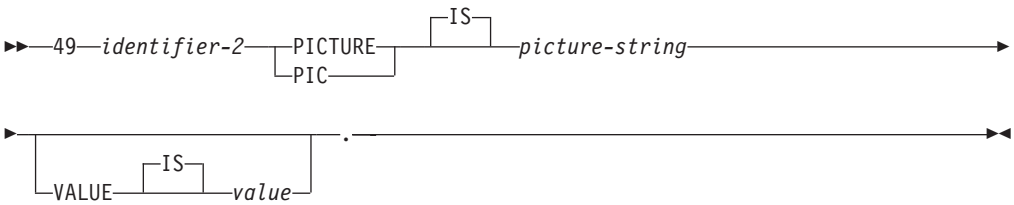
Following is the syntax for character host variables.

Syntax for Character Host Variables in COBOL:



Variable Length





Character Host Variable Consideration:

1. *Picture-string* must have the form *X(m)*. Alternatively, X's may be expanded (for example, ″XXX″ instead of ″X(3)″).
2. *m* is from 1 to 254 for fixed-length strings.
3. *m* is from 1 to 32 700 for variable-length strings.
4. If *m* is greater than 32 672, the host variable will be treated as a LONG VARCHAR string, and its use may be restricted.
5. Use X and 9 as the picture characters in any PICTURE clause. Other characters are not allowed.
6. Variable-length strings consist of a length item and a value item. You can use acceptable COBOL names for the length item and the string item. However, refer to the variable-length string by the collective name in SQL statements.
7. In a CONNECT statement, such as shown below, COBOL character string host variables `dbname` and `userid` will have any trailing blanks removed before processing:

   ```
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

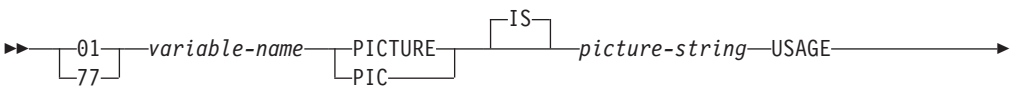
   However, because blanks can be significant in passwords, the `p-word` host variable should be declared as a VARCHAR data item, so that your application can explicitly indicate the significant password length for the CONNECT statement as follows:

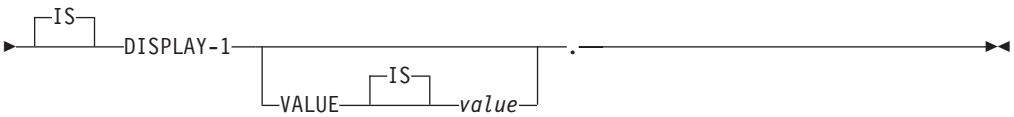
   ```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 dbname PIC X(8).
   01 userid PIC X(8).
   01 p-word.
       49 L PIC S9(4) COMP-5.
       49 D PIC X(18).
   EXEC SQL END DECLARE SECTION END-EXEC.
   PROCEDURE DIVISION.
       MOVE "sample" TO dbname.
       MOVE "userid" TO userid.
       MOVE "password" TO D OF p-word.
       MOVE 8          TO L of p-word.
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

Syntax for Fixed-Length Graphic Host Variables in COBOL

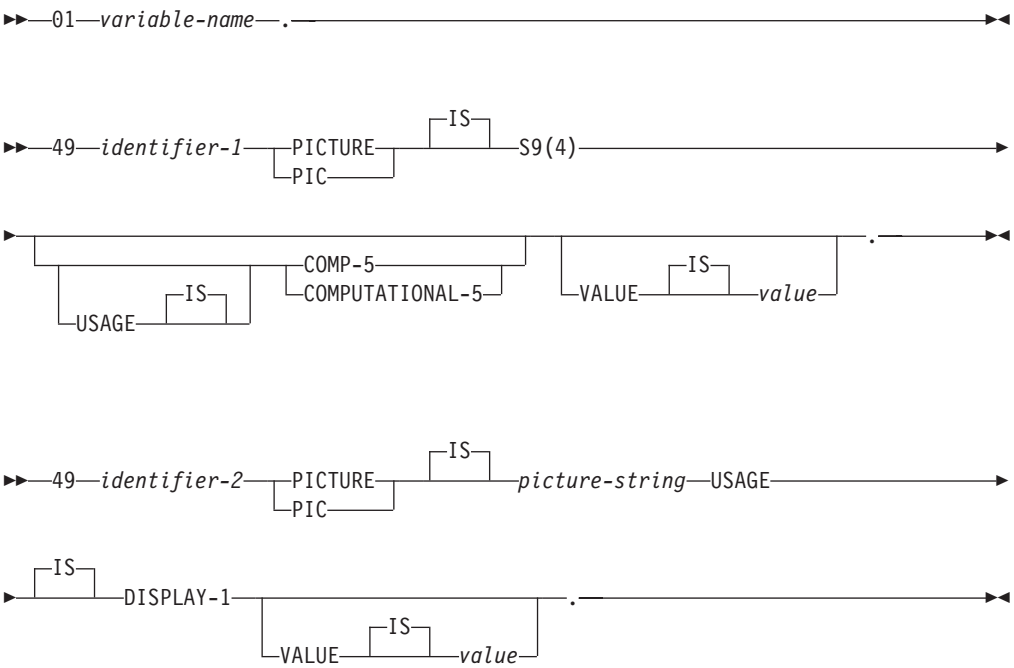
Following is the syntax for graphic host variables.

Syntax for Graphic Host Variables in COBOL:





Variable Length



Graphic Host Variable Considerations:

1. *Picture-string* must have the form *G(m)*. Alternatively, G's may be expanded (for example, "GGG" instead of "G(3)").
2. *m* is from 1 to 127 for fixed-length strings.
3. *m* is from 1 to 16 350 for variable-length strings.
4. If *m* is greater than 16 336, the host variable will be treated as a LONG VARGRAPHIC string, and its use may be restricted.

Indicator Variables in COBOL

Indicator variables should be declared as a `PIC S9(4) COMP-5` data type.

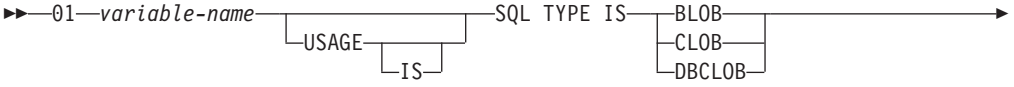
Related concepts:

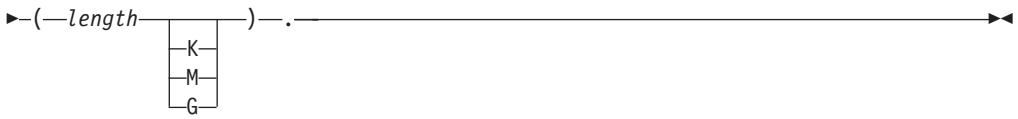
- "Indicator Tables in COBOL" on page 188

Syntax for LOB Host Variables in COBOL

Following is the syntax for declaring large object (LOB) host variables in COBOL.

Syntax for LOB Host Variables in COBOL





LOB Host Variable Considerations:

1. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
2. For DBCLOB 1 <= lob-length <= 1 073 741 823.
3. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in either uppercase, lowercase, or mixed.
4. Initialization within the LOB declaration is not permitted.
5. The host variable name prefixes LENGTH and DATA in the precompiler generated code.

BLOB Example:

Declaring:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

CLOB Example:

Declaring:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

DBCLOB Example:

Declaring:

```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

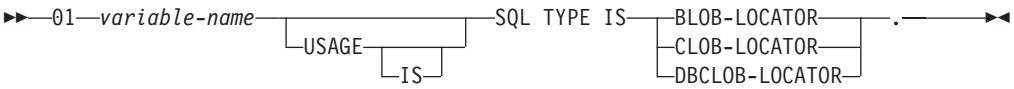
Results in the generation of the following structure:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

Syntax for LOB Locator Host Variables in COBOL

Following is the syntax for declaring large object (LOB) locator host variables in COBOL.

Syntax for LOB Locator Host Variables in COBOL



LOB Locator Host Variable Considerations:

1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR can be either uppercase, lowercase, or mixed.
2. Initialization of locators is not permitted.

BLOB Locator Example (other LOB locator types are similar):

Declaring:

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

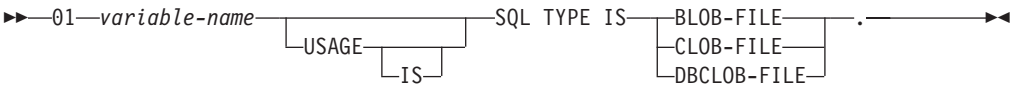
Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

Syntax for File Reference Host Variables in COBOL

Following is the syntax for declaring file reference host variables in COBOL.

Syntax for File Reference Host Variables in COBOL



- SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be either uppercase, lowercase, or mixed.

BLOB File Reference Example (other LOB types are similar):

Declaring:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

Results in the generation of the following declaration:

```
01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

Host Structure Support in COBOL

The COBOL precompiler supports declarations of group data items in the host variable declare section. Among other things, this provides a shorthand for referring to a set of elementary data items in an SQL statement. For example, the following group data item can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
01 staff-record.
    05 staff-id       pic s9(4) comp-5.
    05 staff-name.
        49 l          pic s9(4) comp-5.
```

```
        49 d          pic x(9).
     05 staff-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

Group data items in the declare section can have any of the valid host variable types described above as subordinate data items. This includes all numeric and character types, as well as all large object types. You can nest group data items up to 10 levels. Note that you must declare VARCHAR character types with the subordinate items at level 49, as in the above example. If they are not at level 49, the VARCHAR is treated as a group data item with two subordinates, and is subject to the rules of declaring and using group data items. In the example above, `staff-info` is a group data item, whereas `staff-name` is a VARCHAR. The same principle applies to LONG VARCHAR, VARGRAPHIC, and LONG VARGRAPHIC. You may declare group data items at any level between 02 and 49.

You can use group data items and their subordinates in four ways:

Method 1.

The entire group may be referenced as a single host variable in an SQL statement:

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record
  FROM staff WHERE id = 10 END-EXEC.
```

The precompiler converts the reference to staff-record into a list, separated by commas, of all the subordinate items declared within `staff-record`. Each elementary item is qualified with the group names of all levels to prevent naming conflicts with other items.This is equivalent to the following method.

Method 2.

The second way of using group data items:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-record.staff-id,
  :staff-record.staff-name,
  :staff-record.staff-info.staff-dept,
  :staff-record.staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

Note: The reference to `staff-id` is qualified with its group name using the prefix `staff-record.`, and not `staff-id` of `staff-record` as in pure COBOL.

Assuming there are no other host variables with the same names as the subordinates of `staff-record`, the above statement can also be coded as in method 3, eliminating the explicit group qualification.

Method 3.

Here, subordinate items are referenced in a typical COBOL fashion, without being qualified to their particular group item:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
```

```
:staff-name,
:staff-dept,
:staff-job
FROM staff WHERE id = 10 END-EXEC.
```

As in pure COBOL, this method is acceptable to the precompiler as long as a given subordinate item can be uniquely identified. If, for example, `staff-job` occurs in more than one group, the precompiler issues an error indicating an ambiguous reference:

```
SQL0088N Host variable "staff-job" is ambiguous.
```

Method 4.

To resolve the ambiguous reference, you can use partial qualification of the subordinate item, for example:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-info.staff-dept,
  :staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

Because a reference to a group item alone, as in method 1, is equivalent to a comma-separated list of its subordinates, there are instances where this type of reference leads to an error. For example:

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

Here, the CONNECT statement expects a single character-based host variable. By giving the `staff-record` group data item instead, the host variable results in the following precompile-time error:

```
SQL0087N Host variable "staff-record" is a structure used where
         structure references are not permitted.
```

Other uses of group items that cause an SQL0087N to occur include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables, and SQLDA references. Groups with only one subordinate are permitted in such situations, as are references to individual subordinates, as in methods 2, 3, and 4 above.

Indicator Tables in COBOL

The COBOL precompiler supports the declaration of tables of indicator variables, which are convenient to use with group data items. They are declared as follows:

```
01 <indicator-table-name>.
    05 <indicator-name> pic s9(4) comp-5
                        occurs <table-size> times.
```

For example:

```
01 staff-indicator-table.
    05 staff-indicator pic s9(4) comp-5
                       occurs 7 times.
```

This indicator table can be used effectively with the first format of group item reference above:

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record :staff-indicator
  FROM staff WHERE id = 10 END-EXEC.
```

Here, the precompiler detects that staff-indicator was declared as an indicator table, and expands it into individual indicator references when it processes the SQL statement. staff-indicator(1) is associated with staff-id of staff-record, staff-indicator(2) is associated with staff-name of staff-record, and so on.

Note: If there are k more indicator entries in the indicator table than there are subordinates in the data item (for example, if staff-indicator has 10 entries, making k=6), the k extra entries at the end of the indicator table are ignored. Likewise, if there are k fewer indicator entries than subordinates, the last k subordinates in the group item do not have indicators associated with them. *Note that you can refer to individual elements in an indicator table in an SQL statement*.

Related concepts:

- "Indicator Variables in COBOL" on page 184

REDEFINES in COBOL Group Data Items

You can use the REDEFINES clause when declaring host variables. If you declare a member of a group data item with the REDEFINES clause, and that group data item is referred to as a whole in an SQL statement, any subordinate items containing the REDEFINES clause are not expanded. For example:

```
01 foo.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

Referring to foo in an SQL statement as follows:

```
... INTO :foo ...
```

The above statement is equivalent to:

```
... INTO :foo.a, :foo.b ...
```

That is, the subordinate item a1 that is declared with the REDEFINES clause, is not automatically expanded out in such situations. If a1 is unambiguous, you can explicitly refer to a subordinate with a REDEFINES clause in an SQL statement, as follows:

```
... INTO :foo.a1 ...
```

or

```
... INTO :a1 ...
```

SQL Declare Section with Host Variables for COBOL

The following is a sample SQL declare section with a host variable declared for each supported SQL data type.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
  01 age        PIC S9(4) COMP-5.
  01 divis      PIC S9(9) COMP-5.
  01 salary     PIC S9(6)V9(3) COMP-3.
  01 bonus      USAGE IS COMP-1.
  01 wage       USAGE IS COMP-2.
  01 nm         PIC X(5).
  01 varchar.
     49 leng    PIC S9(4) COMP-5.
```

```
      49 strg    PIC X(14).
   01 longvchar.
      49 len     PIC S9(4) COMP-5.
      49 str     PIC X(6027).
   01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).
   01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.
   01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.
   01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).
   01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.
   01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.
   01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).
   01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.
   01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.
   01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.
   01 dt          PIC X(10).
   01 tm          PIC X(8).
   01 tmstmp      PIC X(26).
   01 wage-ind    PIC S9(4) COMP-5.
 *
 EXEC SQL END DECLARE SECTION END-EXEC.
```

Related reference:

• "Supported SQL Data Types in COBOL" on page 190

Data Type Considerations for COBOL

The sections that follow describe how SQL data types map to COBOL data types.

Supported SQL Data Types in COBOL

Certain predefined COBOL data types correspond to column types. Only these COBOL data types can be declared as host variables.

The following table shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

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

Table 15. 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

Table 15. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
DOUBLE[3] (480 or 481)	01 name USAGE IS COMP-2.	Double-precision floating point
CHAR(*n*) (452 or 453)	01 name PIC X(*n*).	Fixed-length character string
VARCHAR(*n*) (448 or 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=32 672	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*n*<=32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (920 or 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	Identifies BLOB entities residing on the server
BLOB file reference variable[4] (916 or 917)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
DATE (384 or 385)	01 identifier PIC X(10).	10-byte character string
TIME (388 or 389)	01 identifier PIC X(8).	8-byte character string
TIMESTAMP (392 or 393)	01 identifier PIC X(26).	26-byte character string
Note: The following data types are only available in the DBCS environment.		
GRAPHIC(*n*) (468 or 469)	01 name PIC G(*n*) DISPLAY-1.	Fixed-length double-byte character string
VARGRAPHIC(*n*) (464 or 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	Variable length double-byte character string with 2-byte string length indicator
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

Table 15. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
DBCLOB file reference variable[4] (924 or 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following are additional rules for supported COBOL data types:

- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. See the following sample:

  ```
  01 identifier PIC S9(m)V9(n) COMP-3
  ```

  – Use V to denote the decimal point.
  – Values for *n* and *m* must be greater than or equal to 1.
  – The value for $n + m$ cannot exceed 31.
  – The value for *s* equals the value for *n*.
  – The value for *p* equals the value for $n + m$.
  – The repetition factors *(n)* and *(m)* are optional. The following examples are all valid:

  ```
  01 identifier PIC S9(3)V COMP-3
  01 identifier PIC SV9(3) COMP-3
  01 identifier PIC S9V COMP-3
  01 identifier PIC SV9 COMP-3
  ```

  – PACKED-DECIMAL can be used instead of COMP-3.
- Arrays are *not* supported by the COBOL precompiler.

Related concepts:

- "SQL Declare Section with Host Variables for COBOL" on page 189

BINARY/COMP-4 COBOL Data Types

The DB2® COBOL precompiler supports the use of BINARY, COMP, and COMP-4 data types wherever integer host variables and indicators are permitted, as long as the target COBOL compiler views (or can be made to view) the BINARY, COMP, or COMP-4 data types as equivalent to the COMP-5 data type. In this book, such host variables and indicators are shown with the type COMP-5. Target compilers supported by DB2 that treat COMP, COMP-4, BINARY COMP and COMP-5 as equivalent are:

- IBM® COBOL Set for AIX®
- Micro Focus COBOL for AIX

FOR BIT DATA in COBOL

Certain database columns can be declared FOR BIT DATA. These columns, which generally contain characters, are used to hold binary information. The CHAR(*n*), VARCHAR, LONG VARCHAR, and BLOB data types are the COBOL host variable types that can contain binary data. Use these data types when working with columns with the FOR BIT DATA attribute.

Related reference:

- "Supported SQL Data Types in COBOL" on page 190

SQLSTATE and SQLCODE Variables in COBOL

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PICTURE X(5).
01 SQLCODE  PICTURE S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. The `01` can also be `77` and the `PICTURE` can be `PIC`. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications made up of multiple source files, the SQLCODE and SQLSTATE declarations may be included in each source file as shown above.

Japanese or Traditional Chinese EUC, and UCS-2 Considerations for COBOL

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to a the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 because this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

Related concepts:

- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 614

Related reference:

- "VARCHAR scalar function" in the *SQL Reference, Volume 1*
- "VARGRAPHIC scalar function" in the *SQL Reference, Volume 1*

Object Oriented COBOL

If you are using object oriented COBOL, you must observe the following:

- SQL statements can only appear in the first program or class in a compile unit. This restriction exists because the precompiler inserts temporary working data into the first Working-Storage section it sees.
- In an object oriented COBOL program, every class containing SQL statements must have a class-level Working-Storage Section, even if it is empty. This section is used to store data definitions generated by the precompiler.

Chapter 9. Programming in FORTRAN

Programming Considerations for FORTRAN

Special host-language programming considerations are discussed in the following sections. Included is information on language restrictions, host-language-specific include files, embedding SQL statements, host variables, and supported data types for host variables.

Note: FORTRAN support stabilized in DB2 Version 5, and no enhancements for FORTRAN support are planned for the future. For example, the FORTRAN precompiler cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than FORTRAN.

Language Restrictions in FORTRAN

The sections that follow describe the language restrictions for FORTRAN.

Call by Reference in FORTRAN

Some API parameters require addresses rather than values in the call variables. The database manager provides the GET ADDRESS, DEREFERENCE ADDRESS, and COPY MEMORY APIs, which simplify your ability to provide these parameters.

Related reference:

- "sqlgdref - Dereference Address" in the *Administrative API Reference*
- "sqlgaddr - Get Address" in the *Administrative API Reference*
- "sqlgmcpy - Copy Memory" in the *Administrative API Reference*

Debug and Comment Lines in FORTRAN

Some FORTRAN compilers treat lines with a 'D' or 'd' in column 1 as conditional lines. These lines can either be compiled for debugging or treated as comments. The precompiler will always treat lines with a 'D' or 'd' in column 1 as comments.

Precompilation Considerations for FORTRAN

The following items affect the precompiling process:

- The precompiler allows only digits, blanks, and tab characters within columns 1-5 on continuation lines.
- Hollerith constants are not supported in `.sqf` source files.

Multiple-Thread Database Access in FORTRAN

FORTRAN does not support multiple-thread database access.

Input and Output Files for FORTRAN

By default, the input file has an extension of `.sqf`, but if you use the TARGET precompile option the input file can have any extension you prefer.

By default, the output file has an extension of `.f` on UNIX®-based platforms, and `.for` on Windows®-based platforms; however, you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Include Files

The sections that follow describe include files for FORTRAN.

Include Files for FORTRAN

The host-language-specific include files for FORTRAN have the file extension `.f` on UNIX-based platforms, and `.for` on Windows-based platforms. You can use the following FORTRAN include files in your applications.

SQL (sql.f) This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

SQLAPREP (sqlaprep.f)
This file contains definitions required to write your own precompiler.

SQLCA (sqlca_cn.f, sqlca_cs.f)
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

Two SQLCA files are provided for FORTRAN applications. The default, `sqlca_cs.f`, defines the SQLCA structure in an IBM SQL compatible format. The `sqlca_cn.f` file, precompiled with the `SQLCA NONE` option, defines the SQLCA structure for better performance.

SQLCA_92 (sqlca_92.f)
This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of either the `sqlca_cn.f` or the `sqlca_cs.f` files when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.f` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

SQLCODES (sqlcodes.f)
This file defines constants for the SQLCODE field of the SQLCA structure.

SQLDA (sqldact.f)
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

SQLEAU (sqleau.f)
This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

SQLENV (sqlenv.f)
This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

SQLE819A (sqle819a.f)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE819B (sqle819b.f)
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE850A (sqle850a.f)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQLE850B (sqle850b.f)
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLE932A (sqle932a.f)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQLE932B (sqle932b.f)
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

SQL1252A (sql1252a.f)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

SQL1252B (sql1252b.f)
If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

SQLMON (sqlmon.f)
This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

SQLSTATE (sqlstate.f)
This file defines constants for the SQLSTATE field of the SQLCA structure.

SQLUTIL (sqlutil.f)
This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

Related concepts:

• "Include Files in FORTRAN Applications" on page 198

Include Files in FORTRAN Applications

There are two methods for including files: the EXEC SQL INCLUDE statement and the FORTRAN INCLUDE statement. The precompiler will ignore FORTRAN INCLUDE statements, and only process files included with the EXEC SQL statement.

To locate the INCLUDE file, the DB2® FORTRAN precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll`

  If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for `payroll.sqf`, then `payroll.f` (`payroll.for` on Windows®- based platforms) in each directory in which it looks.
- `EXEC SQL INCLUDE 'pay/payroll.f'`

If the file name is enclosed in quotation marks, as above, no extension is added to the name. (For Windows-based platforms, the file would be specified as `'pay\payroll.for'`.)

If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for UNIX®-based platforms, if DB2INCLUDE is set to '`/disk2:myfiles/fortran`', the precompiler searches for '`./pay/payroll.f`', then '`/disk2/pay/payroll.f`', and finally '`./myfiles/cobol/pay/payroll.f`'. The path where the file is actually found is displayed in the precompiler messages. On Windows-based platforms, substitute back slashes (\) for the forward slashes, and substitute `'for'` for the `'f'` extension in the above example.

Note: The setting of DB2INCLUDE is cached by the DB2 command line processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

Related concepts:

- "DB2 registry and environment variables" in the *Administration Guide: Performance*

Related reference:

- "Include Files for FORTRAN" on page 196

Embedded SQL Statements in FORTRAN

Embedded SQL statements consist of the following three elements:

Element	Correct FORTRAN Syntax
Keyword	EXEC SQL
Statement string	Any valid SQL statement with blanks as delimiters
Statement terminator	End of source line.

The end of the source line serves as the statement terminator. If the line is continued, the statement terminator is the end of the last continued line.

For example:

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

The following rules apply to embedded SQL statements:

- Code SQL statements between columns 7 and 72 only.
- Use full-line FORTRAN comments, or SQL comments, but do not use the FORTRAN end-of-line comment '!' character in SQL statements. This comment character may be used elsewhere, including host variable declarations.
- Use blanks as delimiters when coding embedded SQL statements, even though FORTRAN statements do not require blanks as delimiters.
- Use only one SQL statement for each FORTRAN source line. Normal FORTRAN continuation rules apply for statements that require more than one source line. Do not split the EXEC SQL keyword pair between lines.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements.

The format for an SQL comment is a double dash (--), followed by a string of zero or more characters and terminated by a line end.

- FORTRAN comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
  - The extension of using ! to code a FORTRAN comment at the end of a line is not supported within an embedded SQL statement.
- Use exponential notation when specifying a real constant in SQL statements. The database manager interprets a string of digits with a decimal point in an SQL statement as a decimal constant, not a real constant.
- Statement numbers are invalid on SQL statements that precede the first executable FORTRAN statement. If an SQL statement has a statement number associated with it, the precompiler generates a labeled CONTINUE statement that directly precedes the SQL statement.
- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a FORTRAN program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, Windows®-based platforms use the Carriage Return/Line Feed for end-of-line, whereas UNIX®-based platforms use just a Line Feed.

Related reference:

- Appendix A, "Supported SQL Statements," on page 685

Host Variables in FORTRAN

The sections that follow describe how to declare and use host variables in FORTRAN programs.

Host Variables in FORTRAN

Host variables are FORTRAN language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from it. After the application is precompiled, host variables are used by the compiler as any other FORTRAN variable.

Related concepts:

- "Host Variable Names in FORTRAN" on page 201
- "Host Variable Declarations in FORTRAN" on page 201
- "Indicator Variables in FORTRAN" on page 203

Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 202
- "Syntax for Character Host Variables in FORTRAN" on page 202

- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 204
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 205
- "Syntax for File Reference Host Variables in FORTRAN" on page 205

Host Variable Names in FORTRAN

The SQL precompiler identifies host variables by their declared name. The following suggestions apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than `SQL`, `sql`, `DB2`®, or `db2`, which are reserved for system use.

Related concepts:

- "Host Variable Declarations in FORTRAN" on page 201

Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 202
- "Syntax for Character Host Variables in FORTRAN" on page 202
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 204
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 205
- "Syntax for File Reference Host Variables in FORTRAN" on page 205

Host Variable Declarations in FORTRAN

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The FORTRAN precompiler only recognizes a subset of valid FORTRAN declarations as valid host variable declarations. These declarations define either numeric or character variables. A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The programmer must ensure that output variables are long enough to contain the values that they will receive.

Related tasks:

- "Declaring structured type host variables" in the *Application Development Guide: Programming Server Applications*

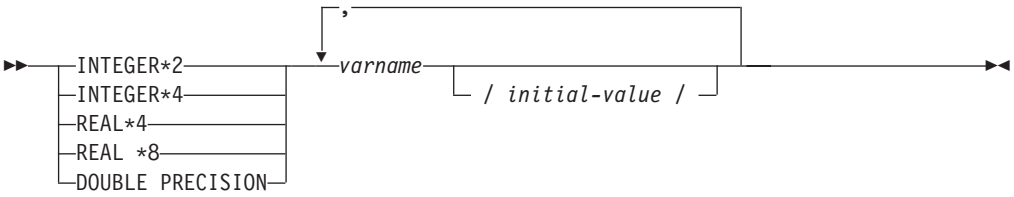
Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 202
- "Syntax for Character Host Variables in FORTRAN" on page 202
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 204
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 205
- "Syntax for File Reference Host Variables in FORTRAN" on page 205

Syntax for Numeric Host Variables in FORTRAN

Following is the syntax for numeric host variables in FORTRAN.

Syntax for Numeric Host Variables in FORTRAN



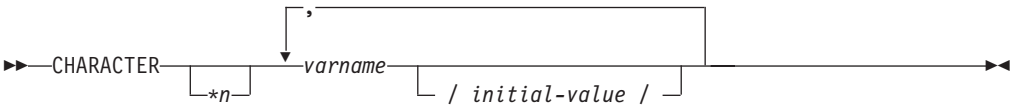
Numeric Host Variable Considerations:

1. REAL*8 and DOUBLE PRECISION are equivalent.
2. Use an E rather than a D as the exponent indicator for REAL*8 constants.

Syntax for Character Host Variables in FORTRAN

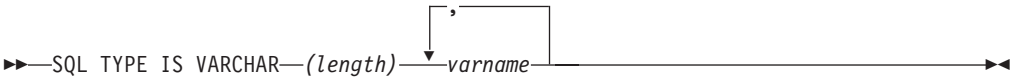
Following is the syntax for fixed-length character host variables.

Syntax for Character Host Variables in FORTRAN: Fixed



Following is the syntax for variable-length character host variables.

Variable Length



Character Host Variable Considerations:

1. **n* has a maximum value of 254.
2. When length is between 1 and 32 672 inclusive, the host variable has type VARCHAR(SQLTYPE 448).
3. When length is between 32 673 and 32 700 inclusive, the host variable has type LONG VARCHAR(SQLTYPE 456).
4. Initialization of VARCHAR and LONG VARCHAR host variables is not permitted within the declaration.

VARCHAR Example:

Declaring:

```
sql type is varchar(1000) my_varchar
```

Results in the generation of the following structure:

```
character    my_varchar(1000+2)
integer*2    my_varchar_length
character    my_varchar_data(1000)
equivalence( my_varchar(1),
+            my_varchar_length )
equivalence( my_varchar(3),
+            my_varchar_data )
```

The application may manipulate both my_varchar_length and my_varchar_data; for example, to set or examine the contents of the host variable. The base name (in this case, my_varchar), is used in SQL statements to refer to the VARCHAR as a whole.

LONG VARCHAR Example:

Declaring:

```
sql type is varchar(10000) my_lvarchar
```

Results in the generation of the following structure:

```
character    my_lvarchar(10000+2)
integer*2    my_lvarchar_length
character    my_lvarchar_data(10000)
equivalence( my_lvarchar(1),
+            my_lvarchar_length )
equivalence( my_lvarchar(3),
+            my_lvarchar_data )
```

The application may manipulate both my_lvarchar_length and my_lvarchar_data; for example, to set or examine the contents of the host variable. The base name (in this case, my_lvarchar), is used in SQL statements to refer to the LONG VARCHAR as a whole.

Note: In a CONNECT statement, such as in the following example, the FORTRAN character string host variables dbname and userid will have any trailing blanks removed before processing.

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

However, because blanks can be significant in passwords, you should declare host variables for passwords as VARCHAR, and have the length field set to reflect the actual password length:

```
EXEC SQL BEGIN DECLARE SECTION
  character*8 dbname, userid
  sql type is varchar(18) passwd
EXEC SQL END DECLARE SECTION
character*18 passwd_string
equivalence(passwd_data,passwd_string)
dbname = 'sample'
userid = 'userid'
passwd_length= 8
passwd_string = 'password'
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

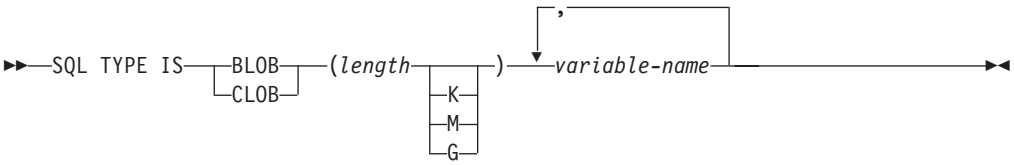
Indicator Variables in FORTRAN

Indicator variables should be declared as an INTEGER*2 data type.

Syntax for Large Object (LOB) Host Variables in FORTRAN

Following is the syntax for declaring large object (LOB) host variables in FORTRAN.

Syntax for Large Object (LOB) Host Variables in FORTRAN



LOB Host Variable Considerations:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB, CLOB, K, M, G can be in either uppercase, lowercase, or mixed.
3. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
4. The initialization of a LOB within a LOB declaration is not permitted.
5. The host variable name prefixes 'length' and 'data' in the precompiler generated code.

BLOB Example:

Declaring:

```
sql type is blob(2m) my_blob
```

Results in the generation of the following structure:

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

CLOB Example:

Declaring:

```
sql type is clob(125m) my_clob
```

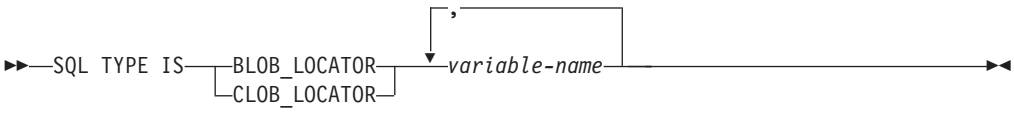
Results in the generation of the following structure:

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

Syntax for Large Object (LOB) Locator Host Variables in FORTRAN

Following is the syntax for declaring large object (LOB) locator host variables in FORTRAN.

Syntax for Large Object (LOB) Locator Host Variables



LOB Locator Host Variable Considerations:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR can be either uppercase, lowercase, or mixed.
3. Initialization of locators is not permitted.

CLOB Locator Example (BLOB locator is similar):

Declaring:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

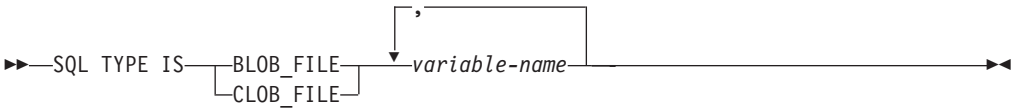
Results in the generation of the following declaration:

```
integer*4 my_locator
```

Syntax for File Reference Host Variables in FORTRAN

Following is the syntax for declaring file reference host variables in FORTRAN.

Syntax for File Reference Host Variables in FORTRAN



File Reference Host Variable Considerations:

1. Graphic types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_FILE, CLOB_FILE can be either uppercase, lowercase, or mixed.

Example of a BLOB file reference variable (CLOB file reference variable is similar):

```
SQL TYPE IS BLOB_FILE my_file
```

Results in the generation of the following declaration:

```
character     my_file(267)
integer*4     my_file_name_length
integer*4     my_file_data_length
integer*4     my_file_file_options
character*255 my_file_name
equivalence(  my_file(1),
```

```
+             my_file_name_length )
 equivalence(  my_file(5),
+             my_file_data_length )
 equivalence(  my_file(9),
+             my_file_file_options )
 equivalence(  my_file(13),
+             my_file_name )
```

SQL Declare Section with Host Variables for FORTRAN

The following is a sample SQL declare section with a host variable declared for each supported data type:

```
EXEC SQL BEGIN DECLARE SECTION
  INTEGER*2     AGE  /26/
  INTEGER*4     DEPT
  REAL*4        BONUS
  REAL*8        SALARY
  CHARACTER     MI
  CHARACTER*112 ADDRESS
  SQL TYPE IS VARCHAR (512) DESCRIPTION
  SQL TYPE IS VARCHAR (32000) COMMENTS
  SQL TYPE IS CLOB (1M) CHAPTER
  SQL TYPE IS CLOB_LOCATOR CHAPLOC
  SQL TYPE IS CLOB_FILE  CHAPFL
  SQL TYPE IS BLOB (1M) VIDEO
  SQL TYPE IS BLOB_LOCATOR VIDLOC
  SQL TYPE IS BLOB_FILE VIDFL
  CHARACTER*10  DATE
  CHARACTER*8   TIME
  CHARACTER*26  TIMESTAMP
  INTEGER*2     WAGE_IND
EXEC SQL END DECLARE SECTION
```

Related reference:

- "Supported SQL Data Types in FORTRAN" on page 206

Supported SQL Data Types in FORTRAN

Certain predefined FORTRAN data types correspond to database manager column types. Only these FORTRAN data types can be declared as host variables.

The following table shows the FORTRAN equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

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

Table 16. SQL Data Types Mapped to FORTRAN Declarations

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
SMALLINT (500 or 501)	INTEGER*2	16-bit, signed integer
INTEGER (496 or 497)	INTEGER*4	32-bit, signed integer
REAL[2] (480 or 481)	REAL*4	Single precision floating point
DOUBLE[3] (480 or 481)	REAL*8	Double precision floating point

Table 16. SQL Data Types Mapped to FORTRAN Declarations (continued)

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
DECIMAL(*p*,*s*) (484 or 485)	No exact equivalent; use REAL*8	Packed decimal
CHAR(*n*) (452 or 453)	CHARACTER**n*	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	SQL TYPE IS VARCHAR(*n*) where *n* is from 1 to 32 672	Variable-length character string
LONG VARCHAR (456 or 457)	SQL TYPE IS VARCHAR(*n*) where *n* is from 32 673 to 32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	SQL TYPE IS CLOB (n) where *n* is from 1 to 2 147 483 647	Large object variable-length character string
CLOB locator variable[4] (964 or 965)	SQL TYPE IS CLOB_LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (920 or 921)	SQL TYPE IS CLOB_FILE	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	SQL TYPE IS BLOB(n) where *n* is from 1 to 2 147 483 647	Large object variable-length binary string
BLOB locator variable[4] (960 or 961)	SQL TYPE IS BLOB_LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable[4] (916 or 917)	SQL TYPE IS BLOB_FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	CHARACTER*10	10-byte character string
TIME (388 or 389)	CHARACTER*8	8-byte character string
TIMESTAMP (392 or 393)	CHARACTER*26	26-byte character string

Notes:

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following is an additional rule for supported FORTRAN data types:

- You may define dynamic SQL statements longer than 254 characters by using VARCHAR, LONG VARCHAR, OR CLOB host variables.

Related concepts:

- "SQL Declare Section with Host Variables for FORTRAN" on page 206

Considerations for Multi-Byte Character Sets in FORTRAN

There are no graphic (multi-byte) host variable data types supported in FORTRAN. Only mixed-character host variables are supported through the `character` data type. It is possible to create a user SQLDA that contains graphic data.

Japanese or Traditional Chinese EUC, and UCS-2 Considerations for FORTRAN

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to a the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 because this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

Related concepts:

- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 614

Related reference:

- "VARCHAR scalar function" in the *SQL Reference, Volume 1*
- "VARGRAPHIC scalar function" in the *SQL Reference, Volume 1*

SQLSTATE and SQLCODE Variables in FORTRAN

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

If neither of these is specified, the `SQLCOD` declaration is assumed during the precompile step. The variable named `SQLSTATE` may also be `SQLSTA`. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications that contain multiple source files, the declarations of `SQLCOD` and `SQLSTATE` may be included in each source file, as shown above.

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Part 3. ADO.NET, OLE DB, and ODBC

Chapter 10. DB2 .NET Data Provider

DB2 .NET Data Provider overview

The DB2® .NET Data Provider is an extension of the ADO.NET interface that allows .NET applications to access a DB2 database through a secure connection, execute commands, and retrieve result sets.

Reference documentation is included with the DB2 .NET Data Provider, presenting detailed information about all the DB2 .NET Data Provider objects and their members. During the DB2 installation process, this documentation is registered with Microsoft® Visual Studio .NET. To view the DB2 .NET Data Provider documentation from Microsoft Visual Studio .NET, select the `Help` menu option, and `Contents`. Once the help viewer opens, filter by *IBM® DB2 .NET Data Provider Help*.

DB2 .NET Data Provider system requirements

The DB2® .NET Data Provider allows your .NET applications to access the following database management systems:

- DB2 Universal Database™ Version 8 for Windows®, UNIX®, and Linux-based computers
- DB2 Universal Database Version 6 (or later) for OS/390® and z/OS™, through DB2 Connect™
- DB2 Universal Database Version 5, Release 1 (or later) for AS/400® and iSeries™, through DB2 Connect
- DB2 Universal Database Version 7.3 (or later) for VSE & VM, through DB2 Connect

Before using the DB2 Install program to install the DB2 .NET Data Provider, you must already have the .NET Framework (Version 1.0 or Version 1.1) installed on the computer. If the .NET Framework is not installed, the DB2 Install program will not install the DB2 .NET Data Provider.

For DB2 Universal Database for AS/400 and iSeries, the following fix is required on the server: APAR ii13348.

Only the .NET Framework Version 1.1 and Visual Studio .NET 2003 are supported for use with DB2 for VSE & VM, and DB2 for iSeries servers. The .NET Framework Version 1.0 and Visual Studio .NET 2002 are not supported for use with these servers.

Programming applications to use the DB2 .NET Data Provider

The following sections describe the main steps in programming a .NET application to access or manipulate data in a DB2 database. Examples in C# and Visual Basic .NET are provided to illustrate each step.

Connecting to a database from an application using the DB2 .NET Data Provider

When using the DB2 .NET Data Provider, a database connection is established through the DB2Connection class. First, you must create a string that stores the connection parameters.

Examples of possible connection strings are:

```
• String connectString = "Database=SAMPLE";
  // When used, attempts to connect to the SAMPLE database.
• String connectString = "Server=srv:50000;Database=SAMPLE;UID=db2adm;PWD=ab1cd";
  // When used, attempts to connect to the SAMPLE database on the server
  // 'srv' through port 50000 using 'db2adm' and 'ab1cd' as the user id and
  // password respectively.
```

To create the database connection, pass the connectString to the DB2Connection constructor. Then use the DB2Connection object's Open method to formally connect to the database identified in connectString.

Connecting to a database in C#:

```
String connectString = "Database=SAMPLE";
DB2Connection conn = new DB2Connection(connectString);
conn.Open();
return conn;
```

Connecting to a database in Visual Basic .NET:

```
Dim connectString As String = "Database=SAMPLE"
Dim conn As DB2Connection = new DB2Connection(connectString)
conn.Open()
Return conn
```

Executing SQL statements from an application using the DB2 .NET Data Provider

When using the DB2 .NET Data Provider, the execution of SQL statements is done through a DB2Command class using its methods ExecuteReader() and ExecuteNonQuery(), and its properties CommandText, CommandType and Transaction. For SQL statements that produce output, the ExecuteReader() method should be used and its results can be retrieved from a DB2DataReader object. For all other SQL statements, the method ExecuteNonQuery() should be used. The Transaction property of the DB2Command object should be initialized to a DB2Transaction. A DB2Transaction object is responsible for rolling back and committing database transactions.

Executing an UPDATE statement in C#:

```
// assume a DB2Connection conn
DB2Command cmd = conn.CreateCommand();
DB2Transaction trans = conn.BeginTransaction();
cmd.Transaction = trans;
cmd.CommandText = "UPDATE staff " +
                  "  SET salary = (SELECT MIN(salary) " +
```

```
                    "                   FROM staff " +
    "                    WHERE id >= 310) " +
                    "  WHERE id = 310";
cmd.ExecuteNonQuery();
```

Executing an UPDATE statement in Visual Basic .NET:

```
' assume a DB2Connection conn
DB2Command cmd = conn.CreateCommand();
DB2Transaction trans = conn.BeginTransaction();
cmd.Transaction = trans;
cmd.CommandText = "UPDATE staff " +
                  "  SET salary = (SELECT MIN(salary) " +
                  "                   FROM staff " +
                  "                   WHERE id >= 310) " +
                  "  WHERE id = 310";
cmd.ExecuteNonQuery();
```

Executing a SELECT statement in C#:

```
// assume a DB2Connection conn
DB2Command cmd = conn.CreateCommand();
DB2Transaction trans = conn.BeginTransaction();
cmd.Transaction = trans;
cmd.CommandText = "SELECT deptnumb, location " +
                  "  FROM org " +
                  "  WHERE deptnumb < 25";
DB2DataReader reader = cmd.ExecuteReader();
```

Executing a SELECT statement in Visual Basic .NET:

```
' assume a DB2Connection conn
Dim cmd As DB2Command = conn.CreateCommand()
Dim trans As DB2Transaction = conn.BeginTransaction()
cmd.Transaction = trans
cmd.CommandText = "UPDATE staff " +
                  "  SET salary = (SELECT MIN(salary) " +
                  "                   FROM staff " +
                  "                   WHERE id >= 310) " +
                  "  WHERE id = 310"
cmd.ExecuteNonQuery()
```

Once your application has performed a database transaction, you must either roll it back or commit it. This is done through the `Commit()` and `Rollback()` methods of a `DB2Transaction` object.

Rolling back or committing a transaction in C#:

```
// assume a DB2Transaction object conn
trans.Rollback();
...
trans.Commit();
```

Rolling back or committing a transaction in C#:

```
' assume a DB2Transaction object conn
trans.Rollback()
...
trans.Commit()
```

Reading result sets from an application using the DB2 .NET Data Provider

When using the DB2 .NET Data Provider, the reading of result sets is done through a `DB2DataReader` object. The `DB2DataReader` method, `Read()` is used to advance to the next row of result set. The methods `GetString()`, `GetInt32()`, `GetDecimal()`,

and other methods for all the available data types are used to extract data from the individual columns of output. DB2DataReader's Close() method is used to close the DB2DataReader, which should always be done when finished reading output.

Reading a result set in C#:

```
// assume a DB2DataReader reader
Int16 deptnum = 0;
String location="";

// Output the results of the query
while(reader.Read())
{
  deptnum = reader.GetInt16(0);
  location = reader.GetString(1);
  Console.WriteLine("    " + deptnum + " " + location);
}
reader.Close();
```

Reading a result set in Visual Basic .NET:

```
' assume a DB2DataReader reader
Dim deptnum As Int16 = 0
Dim location As String ""

' Output the results of the query
Do While (reader.Read())
  deptnum = reader.GetInt16(0)
  location = reader.GetString(1)
  Console.WriteLine("    " & deptnum & " " & location)
Loop
reader.Close();
```

Calling stored procedures from an application using the DB2 .NET Data Provider

When using the DB2 .NET Data Provider, you can call stored procedures by using a DB2Command object. The default value of the CommandType property is CommandType.Text. This is the appropriate value for SQL statements and can also be used to call stored procedures. However, calling stored procedures is easier when you set CommandType to CommandType.StoredProcedure. In this case, you only need to specify the stored procedure name and any parameters.

The following examples demonstrates how to invoke a stored procedure called INOUT_PARAM, with the CommandType property set to either CommandType.StoredProcedure or CommandType.Text.

Calling a stored procedure by setting the CommandType property of the DB2Command to CommandType.StoredProcedure in C#:

```
// assume a DB2Connection comm
DB2Transaction trans = conn.BeginTransaction();
DB2Command cmd = conn.CreateCommand();
String procName = "INOUT_PARAM";
cmd.Transaction = trans;
cmd.CommandType = CommandType.StoredProcedure;
cmd.CommandText = procName;

// Register input-output and output parameters for the DB2Command
...

// Call the stored procedure
Console.WriteLine("  Call stored procedure named " + procName);
cmd.ExecuteNonQuery();
```

Calling a stored procedure by setting the CommandType property of the DB2Command to CommandType.Text in C#:

```
// assume a DB2Connection comm
DB2Transaction trans = conn.BeginTransaction();
DB2Command cmd = conn.CreateCommand();
String procName = "INOUT_PARAM";
String procCall = "CALL INOUT_PARAM (?, ?, ?)";
cmd.Transaction = trans;
cmd.CommandType = CommandType.Text;
cmd.CommandText = procCall;

// Register input-output and output parameters for the DB2Command
...

// Call the stored procedure
Console.WriteLine("  Call stored procedure named " + procName);
cmd.ExecuteNonQuery();
```

Calling a stored procedure by setting the CommandType property of the DB2Command to CommandType.StoredProcedure in Visual Basic .NET:

```
' assume a DB2DataReader reader
Dim trans As DB2Transaction = conn.BeginTransaction()
Dim cmd As DB2Command = conn.CreateCommand()
Dim procName As String = "INOUT_PARAM"
cmd.Transaction = trans
cmd.CommandType = CommandType.StoredProcedure
cmd.CommandText = procName

' Register input-output and output parameters for the DB2Command
...

' Call the stored procedure
Console.WriteLine("  Call stored procedure named " & procName)
cmd.ExecuteNonQuery()
```

Calling a stored procedure by setting the CommandType property of the DB2Command to CommandType.Text in Visual Basic .NET:

```
' assume a DB2DataReader reader
Dim trans As DB2Transaction = conn.BeginTransaction()
Dim cmd As DB2Command = conn.CreateCommand()
Dim procName As String = "INOUT_PARAM"
Dim procCall As String = "CALL INOUT_PARAM (?, ?, ?)"
cmd.Transaction = trans
cmd.CommandType = CommandType.Text
cmd.CommandText = procCall

' Register input-output and output parameters for the DB2Command
...

' Call the stored procedure
Console.WriteLine("  Call stored procedure named " & procName)
cmd.ExecuteNonQuery()
```

Supported SQL data types for the DB2 .NET Data Provider

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

Table 17. Mapping DB2 Data Types to .NET data types

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

Notes:

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

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

- Parameters in CLR routines

Related concepts:

- "Parameter styles for external routines" in the *Application Development Guide: Programming Server Applications*
- "Common language runtime (CLR) routines" in the *Application Development Guide: Programming Server Applications*
- "Parameters in CLR routines" in the *Application Development Guide: Programming Server Applications*

Related tasks:

- "Passing structured type parameters to external routines" in the *Application Development Guide: Programming Server Applications*
- "Creating CLR routines" in the *Application Development Guide: Programming Server Applications*
- "Examples of CLR user-defined functions in C#" in the *Application Development Guide: Programming Server Applications*
- "Examples of CLR procedures in C#" in the *Application Development Guide: Programming Server Applications*

Related samples:

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

Chapter 11. IBM OLE DB Provider for DB2

Purpose of the IBM OLE DB Provider for DB2

Microsoft® OLE DB is a set of OLE/COM interfaces that provides applications with uniform access to data stored in diverse information sources. The OLE DB architecture defines OLE DB consumers and OLE DB providers. An OLE DB consumer is any system or application that uses OLE DB interfaces; an OLE DB provider is a component that exposes OLE DB interfaces.

The IBM® OLE DB Provider for DB2® allows DB2 to act as a resource manager for the OLE DB provider. This support gives OLE DB-based applications the ability to extract or query DB2 data using the OLE interface. The IBM OLE DB Provider for DB2, whose provider name is IBMDADB2, enables OLE DB consumers to access data on a DB2 Universal Database™ server. If DB2 Connect™ is installed, these OLE DB consumers can also access data on a host DBMS such as DB2 for MVS™, DB2 for VM/VSE, or SQL/400.

The IBM OLE DB Provider for DB2 offers the following features:

- Support level 0 of the OLE DB provider specification, including some additional level 1 interfaces.
- A free threaded provider implementation, which enables the application to create components in one thread and use those components in any other thread.
- An Error Lookup Service that returns DB2 error messages.

Note that the IBM OLE DB Provider resides on the client and is different from the OLE DB table functions, which are also supported by DB2 UDB.

Subsequent sections of this document describe the specific implementation of the IBM OLE DB Provider for DB2. For more information on the Microsoft OLE DB 2.0

specification, refer to the Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK, available from Microsoft Press.

Version Compliance:

The IBM OLE DB Provider for DB2 complies with Version 2.7 of the Microsoft OLE DB specification.

System Requirements:

Refer to the announcement letter for the IBM OLE DB Provider for DB2 Servers to see the supported Windows® operating systems.

To install the IBM OLE DB Provider for DB2, you must first be running on one of the supported operating systems listed above. You also need to install the DB2 Application Development Client, as well as the Microsoft Data Access Components (MDAC) Version 2.7 or higher, which was available at the time of writing from the following site: `http://www.microsoft.com/data`.

Related reference:

- "IBM OLE DB Provider Support for OLE DB Components and Interfaces" on page 227

Application Types Supported by the IBM OLE DB Provider for DB2

With the IBM® OLE DB Provider for DB2®, you can create the following types of applications:

- ADO applications, including:
  - Microsoft® Visual Studio C++ applications
  - Microsoft Visual Basic applications
- ADO.NET applications using the OLE DB .NET Data Provider
- C/C++ applications which access IBMDADB2 directly using the OLE DB interfaces, including ATL applications whose Data Access Consumer Objects were generated by the ATL COM AppWizard.

OLE DB Services

The sections that follow describe OLE DB services.

Thread Model Supported by IBM OLE DB Provider

The IBM® OLE DB Provider for DB2® supports the Free Threaded model, which allows applications to create components in one thread and use those components in any other thread.

Large Object Manipulation with the IBM OLE DB Provider

To get and set data as storage objects (DBTYPE_IUNKNOWN) with IBMDADB2, use the ISequentialStream interface as follows:

- To bind a storage object to a parameter, the DBOBJECT in the DBBINDING structure can only contain the value STGM_READ for the dwFlag field. IBMDADB2 will execute the Read method of the ISequentialStream interface of the bound object.

- To get data from a storage object, your application must perform a Read method on the ISequentialStream interface of the storage object.
- When getting data, the value of the length part is the length of the real data, not the length of the IUnknown pointer.

Schema Rowsets Supported by the IBM OLE DB Provider

The following table shows the schema rowsets that are supported by `IDBSchemaRowset`. Note that unsupported columns will be set to null in the rowsets.

Table 18. Schema Rowsets Supported by the IBM OLE DB Provider for DB2

Supported GUIDs	Supported Restrictions	Supported Columns	Notes
DBSCHEMA _COLUMN_PRIVILEGES	COLUMN_NAME TABLE_NAME TABLE_SCHEMA	COLUMN_NAME GRANTEE GRANTOR IS_GRANTABLE PRIVILEGE_TYPE TABLE_NAME TABLE_SCHEMA	
DB_SCHEMA_COLUMNS	COLUMN_NAME TABLE_NAME TABLE_SCHEMA	CHARACTER_MAXIMUM_LENGTH CHARACTER_OCTET_LENGTH COLUMN_DEFAULT COLUMN_FLAGS COLUMN_HASDEFAULT COLUMN_NAME DATA_TYPE DESCRIPTION IS_NULLABLE NUMERIC_PRECISION NUMERIC_SCALE ORDINAL_POSITION TABLE_NAME TABLE_SCHEMA	
DBSCHEMA_FOREIGN_KEYS	FK_TABLE_NAME FK_TABLE_SCHEMA PK_TABLE_NAME PK_TABLE_SCHEMA	DEFERRABILITY DELETE_RULE FK_COLUMN_NAME FK_NAME FK_TABLE_NAME FK_TABLE_SCHEMA ORDINAL PK_COLUMN_NAME PK_NAME PK_TABLE_NAME PK_TABLE_SCHEMA UPDATE_RULE	Must specify at least one of the following restrictions: PK_TABLE_NAME or FK_TABLE_NAME No “%” wildcard allowed.
DBSCHEMA_INDEXES	TABLE_NAME TABLE_SCHEMA	CARDINALITY CLUSTERED COLLATION COLUMN_NAME INDEX_NAME INDEX_SCHEMA ORDINAL_POSITION PAGES TABLE_NAME TABLE_SCHEMA TYPE UNIQUE	No sort order supported. Sort order, if specified, will be ignored.
DBSCHEMA_PRIMARY_KEYS	TABLE_NAME TABLE_SCHEMA	COLUMN_NAME ORDINAL PK_NAME TABLE_NAME TABLE_SCHEMA	Must specify at least the following restrictions: TABLE_NAME No “%” wildcard allowed.

Table 18. Schema Rowsets Supported by the IBM OLE DB Provider for DB2 (continued)

Supported GUIDs	Supported Restrictions	Supported Columns	Notes
DBSCHEMA _PROCEDURE_PARAMETERS	PARAMETER_NAME PROCEDURE_NAME PROCEDURE_SCHEMA	CHARACTER_MAXIMUM_LENGTH CHARACTER_OCTET_LENGTH DATA_TYPE DESCRIPTION IS_NULLABLE NUMERIC_PRECISION NUMERIC_SCALE ORDINAL_POSITION PARAMETER_DEFAULT PARAMETER_HASDEFAULT PARAMETER_NAME PARAMETER_TYPE PROCEDURE_NAME PROCEDURE_SCHEMA TYPE_NAME	
DBSCHEMA_PROCEDURES	PROCEDURE_NAME PROCEDURE_SCHEMA	DESCRIPTION PROCEDURE_NAME PROCEDURE_SCHEMA PROCEDURE_TYPE	
DBSCHEMA_PROVIDER_TYPES	DATA_TYPE BEST_MATCH	AUTO_UNIQUE_VALUE BEST_MATCH CASE_SENSITIVE CREATE_PARAMS COLUMN_SIZE DATA_TYPE FIXED_PREC_SCALE IS_FIXEDLENGTH IS_LONG IS_NULLABLE LITERAL_PREFIX LITERAL_SUFFIX LOCAL_TYPE_NAME MINIMUM_SCALE MAXIMUM_SCALE SEARCHABLE TYPE_NAME UNSIGNED_ATTRIBUTE	
DBSCHEMA_STATISTICS	TABLE_NAME TABLE_SCHEMA	CARDINALITY TABLE_NAME TABLE_SCHEMA	No sort order supported. Sort order, if specified, will be ignored.
DBSCHEMA _TABLE_PRIVILEGES	TABLE_NAME TABLE_SCHEMA	GRANTEE GRANTOR IS_GRANTABLE PRIVILEGE_TYPE TABLE_NAME TABLE_SCHEMA	
DBSCHEMA_TABLES	TABLE_NAME TABLE_SCHEMA TABLE_TYPE	DESCRIPTION TABLE_NAME TABLE_SCHEMA TABLE_TYPE	

OLE DB Services Automatically Enabled by IBM OLE DB Provider

By default, the IBM® OLE DB Provider for DB2® automatically enables all the OLE DB services by adding a registry entry OLEDB_SERVICES under the class ID (CLSID) of the provider with the DWORD value of 0xFFFFFFFF. The meaning of this value is as follows:

Table 19. OLE DB Services

Enabled Services	DWORD Value
All services (default)	0xFFFFFFFF
All except pooling and AutoEnlistment	0xFFFFFFFC

Table 19. OLE DB Services (continued)

Enabled Services	DWORD Value
All except client cursor	0xFFFFFFFB
All except pooling, enlistment and cursor	0xFFFFFFF8
No services	0x000000000

Data Services

The sections that follow describe data services considerations.

Supported Cursor Modes for the IBM OLE DB Provider

The IBM® OLE DB Provider for DB2® natively supports read-only, forward-only, read-only scrollable, and updatable scrollable cursors.

Data Type Mappings between DB2 and OLE DB

The IBM OLE DB Provider supports data type mappings between DB2 data types and OLE DB data types. The following table provides a complete list of supported mappings and available names for indicating the data types of columns and parameters.

Table 20. Data Type Mappings between DB2 Data Types and OLE DB Data Types

DB2 Data Types	OLE DB Data Types Indicators	OLE DB Standard Type Names	DB2 Specific Names
SMALLINT	DBTYPE_I2	"DBTYPE_I2"	"SMALLINT"
INTEGER	DBTYPE_I4	"DBTYPE_I4"	"INTEGER" or "INT"
BIGINT	DBTYPE_I8	"DBTYPE_I8"	"BIGINT"
REAL	DBTYPE_R4	"DBTYPE_R4"	"REAL"
FLOAT	DBTYPE_R8	"DBTYPE_R8"	"FLOAT"
DOUBLE	DBTYPE_R8	"DBTYPE_R8"	"DOUBLE" or "DOUBLE PRECISION"
DECIMAL	DBTYPE_NUMERIC	"DBTYPE_NUMERIC"	"DEC" or "DECIMAL"
NUMERIC	DBTYPE_NUMERIC	"DBTYPE_NUMERIC"	"NUM" or "NUMERIC"
DATE	DBTYPE_DBDATE	"DBTYPE_DBDATE"	"DATE"
TIME	DBTYPE_DBTIME	"DBTYPE_DBTIME"	"TIME"
TIMESTAMP	DBTYPE_DBTIMESTAMP	"DBTYPE_DBTIMESTAMP"	"TIMESTAMP"
CHAR	DBTYPE_STR	"DBTYPE_CHAR"	"CHAR" or "CHARACTER"
VARCHAR	DBTYPE_STR	"DBTYPE_VARCHAR"	"VARCHAR"
LONG VARCHAR	DBTYPE_STR	"DBTYPE_LONGVARCHAR"	"LONG VARCHAR"
CLOB	DBTYPE_STR and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"DBTYPE_CHAR" "DBTYPE_VARCHAR" "DBTYPE_LONGVARCHAR" and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"CLOB"
GRAPHIC	DBTYPE_WSTR	"DBTYPE_WCHAR"	"GRAPHIC"
VARGRAPHIC	DBTYPE_WSTR	"DBTYPE_WVARCHAR"	"VARGRAPHIC"
LONG VARGRAPHIC	DBTYPE_WSTR	"DBTYPE_WLONGVARCHAR"	"LONG VARGRAPHIC"

Table 20. Data Type Mappings between DB2 Data Types and OLE DB Data Types (continued)

DB2 Data Types	OLE DB Data Types Indicators	OLE DB Standard Type Names	DB2 Specific Names
DBCLOB	DBTYPE_WSTR and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"DBTYPE_WCHAR" "DBTYPE_WVARCHAR" "DBTYPE_WLONGVARCHAR" and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"DBCLOB"
CHAR(n) FOR BIT DATA	DBTYPE_BYTES	"DBTYPE_BINARY"	
VARCHAR(n) FOR BIT DATA	DBTYPE_BYTES	"DBTYPE_VARBINARY"	
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES	"DBTYPE_LONGVARBINARY"	
BLOB	DBTYPE_BYTES and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"DBTYPE_BINARY" "DBTYPE_VARBINARY" "DBTYPE_LONGVARBINARY" and DBCOLUMNFLAGS_ISLONG or DBPARAMFLAGS_ISLONG	"BLOB"
DATA LINK	DBTYPE_STR	"DBTYPE_CHAR"	"DATA LINK"

Data Conversion for Setting Data from OLE DB Types to DB2 Types

The IBM OLE DB Provider supports data conversions for setting data from OLE DB types to DB2 types. Note that truncation of the data may occur in some cases, depending on the types and the value of the data.

Table 21. Data Conversions from OLE DB Types to DB2 Types

	DB2 Data Types																						
																			For Bit Data				
OLE DB Type Indicator	SMALLINT	INTEGER	BIGINT	REAL	FLOAT DOUBLE	DECIMAL NUMERIC	DATE	TIME	TIMESTAMP	CHAR	VARCHAR	LONG VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC	DBCLOB	CHAR	VARCHAR	LONG VARCHAR	BLOB	DATALINK
DBTYPE_EMPTY																							
DBTYPE_NULL																							
DBTYPE_RESERVED																							
DBTYPE_I1	X	X	X	X	X	X				X	X												
DBTYPE_I2	X	X	X	X	X	X				X	X												
DBTYPE_I4	X	X	X	X	X	X				X	X												
DBTYPE_I8	X	X	X	X	X	X				X	X												
DBTYPE_UI1	X	X	X	X	X	X				X	X												
DBTYPE_UI2	X	X	X	X	X	X				X	X												
DBTYPE_UI4	X	X	X	X	X	X				X	X												

Table 21. Data Conversions from OLE DB Types to DB2 Types (continued)

OLE DB Type Indicator	DB2 Data Types																						
																		For Bit Data					
	SMALLINT	INTEGER	BIGINT	REAL	FLOAT DOUBLE	DECIMAL NUMERIC	DATE	TIME	TIMESTAMP	CHAR	VARCHAR	LONG VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC	DBCLOB	CHAR	VARCHAR	LONG VARCHAR	BLOB	DATALINK
DBTYPE_UI8	X	X	X	X	X	X				X	X												
DBTYPE_R4	X	X	X	X	X	X				X	X												
DBTYPE_R8	X	X	X	X	X	X				X	X												
DBTYPE_CY																							
DBTYPE_DECIMAL	X	X	X	X	X	X				X	X												
DBTYPE_NUMERIC	X	X	X	X	X	X				X	X												
DBTYPE_DATE																							
DBTYPE_BOOL	X	X	X	X	X	X				X	X												
DBTYPE_BYTES			X			X				X	X	X				X		X	X	X			
DBTYPE_BSTR – to be determined																							
DBTYPE_STR	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X	
DBTYPE_WSTR														X	X	X							
DBTYPE_VARIANT – to be determined																							
DBTYPE_IDISPATCH																							
DBTYPE_IUNKNOWN										X	X	X	X	X	X	X	X	X	X	X			
DBTYPE_GUID																							
DBTYPE_ERROR																							
DBTYPE_BYREF																							
DBTYPE_ARRAY																							
DBTYPE_VECTOR																							
DBTYPE_UDT																							
DBTYPE_DBDATE							X		X	X	X												
DBTYPE_DBTIME								X	X	X	X												
DBTYPE_DBTIMESTAMP							X	X	X	X	X												
DBTYPE_FILETIME																							
DBTYPE_PROP_VARIANT																							
DBTYPE_HCHAPTER																							
DBTYPE_VARNUMERIC																							

Related reference:

- "Data Conversion for Setting Data from DB2 Types to OLE DB Types" on page 226

Data Conversion for Setting Data from DB2 Types to OLE DB Types

For getting data, the IBM OLE DB Provider allows data conversions from DB2 types to OLE DB types. Note that truncation of the data may occur in some cases, depending on the types and the value of the data.

Table 22. Data Conversions from DB2 Types to OLE DB Types

	DB2 Data Types																					
																			For Bit Data			
OLE DB Type Indicator	SMALLINT	INTEGER	BIGINT	REAL	FLOAT DOUBLE	DECIMAL NUMERIC	DATE	TIME	TIMESTAMP	CHAR	VARCHAR	LONG VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC	DBCLOB	CHAR	VARCHAR	LONG VARCHAR	BLOB	DATALINK
DBTYPE_EMPTY																						
DBTYPE_NULL																						
DBTYPE_RESERVED																						
DBTYPE_I1	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_I2	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_I4	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_I8	X	X	X	X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_UI1	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_UI2	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_UI4	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_UI8	X	X	X	X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_R4	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_R8	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_CY	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_DECIMAL	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_NUMERIC	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_DATE	X	X		X	X		X	X	X	X	X	X		X	X	X						X
DBTYPE_BOOL	X	X		X	X	X				X	X	X		X	X	X		X	X	X		X
DBTYPE_BYTES	X	X		X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_BSTR	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_STR	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_WSTR	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_VARIANT	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_IDISPATCH																						
DBTYPE_IUNKNOWN	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DBTYPE_GUID										X	X	X		X	X	X		X	X	X		X
DBTYPE_ERROR																						
DBTYPE_BYREF																						

Table 22. Data Conversions from DB2 Types to OLE DB Types (continued)

OLE DB Type Indicator	DB2 Data Types																					
	SMALLINT	INTEGER	BIGINT	REAL	FLOAT DOUBLE	DECIMAL NUMERIC	DATE	TIME	TIMESTAMP	CHAR	VARCHAR	LONG VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC	DBCLOB	For Bit Data			BLOB	DATALINK
																		CHAR	VARCHAR	LONG VARCHAR		
DBTYPE_ARRAY																						
DBTYPE_VECTOR																						
DBTYPE_UDT																						
DBTYPE_DBDATE							X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_DBTIME							X	X	X	X	X	X		X	X	X						X
DBTYPE_DBTIMESTAMP							X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_FILETIME			X				X	X	X	X	X	X		X	X	X		X	X	X		X
DBTYPE_PROP_VARIANT	X	X	X	X	X					X	X	X		X	X	X		X	X	X		X
DBTYPE_HCHAPTER																						
DBTYPE_VARNUMERIC																						

Note: When the application performs the ISequentialStream::Read to get the data from the storage object, the format of the data returned depends on the column data type:

- For non character and binary data types, the data of the column is exposed as a sequence of bytes which represent those values in the operating system.
- For character data type, the data is first converted to DBTYPE_STR.
- For DBCLOB, the data is first converted to DBTYPE_WCHAR.

Related reference:

- "Data Conversion for Setting Data from OLE DB Types to DB2 Types" on page 224

IBM OLE DB Provider Restrictions

Following are the restrictions for the IBM® OLE DB Provider:

- IBMDADB2 supports auto commit and user-controlled transaction scope with the `ITransactionLocal` interface. Auto commit transaction scope is the default scope. Nested transactions are not supported.
- `RestartPosition` is not supported when the command text contains parameters.
- IBMDADB2 does not quote table names passed through the DBID parameters, which are parameters used by the `IOpenRowset` interface. Instead, the OLE DB consumer must add quotes to the table names when quotes are required.

IBM OLE DB Provider Support for OLE DB Components and Interfaces

The following table lists the OLE DB components and interfaces that are supported by the IBM OLE DB Provider and the Microsoft OLE DB Provider for ODBC.

Table 23. Comparison of OLE DB Components and Interfaces Supported by the IBM OLE DB Provider for DB2 and the Microsoft OLE DB Provider for ODBC

	Interface	DB2	ODBC Provider
BLOB			
	ISequentialStream	Yes	Yes
Command			
	IAccessor	Yes	Yes
	ICommand	Yes	Yes
	ICommandPersist	No	No
	ICommandPrepare	Yes	Yes
	ICommandProperties	Yes	Yes
	ICommandText	Yes	Yes
	ICommandWithParameters	Yes	Yes
	IColumnsInfo	Yes	Yes
	IColumnsRowset	Yes	Yes
	IConvertType	Yes	Yes
	ISupportErrorInfo	Yes	Yes
DataSource			
	IConnectionPoint	No	Yes
	IDBAsynchNotify (consumer)	No	No
	IDBAsynchStatus	No	No
	IDBConnectionPointContainer	No	Yes
	IDBCreateSession	Yes	Yes
	IDBDataSourceAdmin	No	No
	IDBInfo	Yes	Yes
	IDBInitialize	Yes	Yes
	IDBProperties	Yes	Yes
	IPersist	Yes	No
	IPersistFile	Yes	Yes
	ISupportErrorInfo	Yes	Yes
Enumerator			
	IDBInitialize	Yes	Yes
	IDBProperties	Yes	Yes
	IParseDisplayName	Yes	No
	ISourcesRowset	Yes	Yes
	ISupportErrorInfo	Yes	Yes
Error Lookup Service			
	IErrorLookUp	Yes	Yes
Error Object			
	IErrorInfo	Yes	No
	IErrorRecords	Yes	No
	ISQLErrorInfo (custom)	Yes	No

Table 23. Comparison of OLE DB Components and Interfaces Supported by the IBM OLE DB Provider for DB2 and the Microsoft OLE DB Provider for ODBC (continued)

	Interface	DB2	ODBC Provider
Multiple Results			
	IMultipleResults	Yes	Yes
	ISupportErrorInfo	Yes	Yes
RowSet			
	IAccessor	Yes	Yes
	IColumnsRowset	Yes	Yes
	IColumnsInfo	Yes	Yes
	IConvertType	Yes	Yes
	IChapteredRowset	No	No
	IConnectionPointContainer	Yes	Yes
	IDBAsynchStatus	No	No
	IParentRowset	No	No
	IRowset	Yes	Yes
	IRowsetChange	Yes	Yes
	IRowsetChapterMember	No	No
	IRowsetFind	No	No
	IRowsetIdentity	Yes	Yes
	IRowsetIndex	No	No
	IRowsetInfo	Yes	Yes
	IRowsetLocate	Yes	Yes
	IRowsetNotify (consumer)	Yes	No
	IRowsetRefresh	Cursor Service Component	Yes
	IRowsetResynch	Cursor Service Component	Yes
	IRowsetScroll	Yes[1]	Yes
	IRowsetUpdate	Cursor Service Component	Yes
	IRowsetView	No	No
	ISupportErrorInfo	Yes	Yes
Notes: 1. The values to be returned are approximations. Deleted rows will not be skipped.			
Session			
	IAlterIndex	No	No
	IAlterTable	No	No
	IDBCreateCommand	Yes	Yes
	IDBSchemaRowset	Yes	Yes
	IGetDataSource	Yes	Yes
	IIndexDefinition	No	No
	IOpenRowset	Yes	Yes
	ISessionProperties	Yes	Yes
	ISupportErrorInfo	Yes	Yes

Table 23. Comparison of OLE DB Components and Interfaces Supported by the IBM OLE DB Provider for DB2 and the Microsoft OLE DB Provider for ODBC (continued)

	Interface	DB2	ODBC Provider
	ITableDefinition	No	No
	ITableDefinitionWithConstraints	No	No
	ITransaction	Yes	Yes
	ITransactionJoin	Yes	Yes
	ITransactionLocal	Yes	Yes
	ITransactionObject	No	No
	ITransactionOptions	No	Yes
View Objects			
	IViewChapter	No	No
	IViewFilter	No	No
	IViewRowset	No	No
	IViewSort	No	No

IBM OLE DB Provider support for OLE DB properties

The following table shows the OLE DB properties that are supported by the IBM OLE DB Provider:

Table 24. Properties Supported by the IBM OLE DB Provider for DB2

Property Group	Property Set	Properties	Default Value	R/W
Data Source	DBPROPSET_DATASOURCE	DBPROP_MULTIPLECONNECTIONS	VARIANT_FALSE	R
		DBPROP_RESETDATASOURCE	DBPROPVAL_RD_RESETALL	R/W
Data Source Information	DBPROPSET _DATASOURCEINFO	DBPROP_ACTIVESESSIONS	0	R
		DBPROP_ASYNCTXNABORT	VARIANT_FALSE	R
		DBPROP_ASYNCTXNCOMMIT	VARIANT_FALSE	R
		DBPROP_BYREFACCESSORS	VARIANT_FALSE	R
		DBPROP_COLUMNDEFINITION	DBPROPVAL_CD_NOTNULL	R
		DBPROP_CONCATNULLBEHAVIOR	DBPROPVAL_CB_NULL	R
		DBPROP_CONNECTIONSTATUS	DBPROPVAL_CS_INITIALIZED	R
		DBPROP_DATASOURCENAME	N/A	R
		DBPROP_DATASOURCEREADONLY	VARIANT_FALSE	R
		DBPROP_DBMSNAME	N/A	R
		DBPROP_DBMSVER	N/A	R
		DBPROP_DSOTHREADMODEL	DBPROPVAL_RT_FREETHREAD	R
		DBPROP_GROUPBY	DBPROPVAL_GB_CONTAINS_SELECT	R
		DBPROP_IDENTIFIERCASE	DBPROPVAL_IC_UPPER	R
		DBPROP_MAXINDEXSIZE	0	R
		DBPROP_MAXROWSIZE	0	R
		DBPROP_MAXROWSIZEINCLUDESBLOB	VARIANT_TRUE	R
		DBPROP_MAXTABLEINSELECT	0	R
		DBPROP_MULTIPLEPARAMSETS	VARIANT_FALSE	R
		DBPROP_MULTIPLERESULTS	DBPROPVAL_MR_SUPPORTED	R
		DBPROP_MULTIPLESTORAGEOBJECTS	VARIANT_TRUE	R
		DBPROP_MULTITABLEUPDATE	VARIANT_FALSE	R
		DBPROP_NULLCOLLATION	DBPROPVAL_NC_LOW	R
		DBPROP_OLEOBJECTS	DBPROPVAL_OO_BLOB	R
		DBPROP_ORDERBYCOLUMNSINSELECT	VARIANT_FALSE	R
		DBPROP _OUTPUTPARAMETERAVAILABILITY	DBPROPVAL_OA_ATEXECUTE	R

Table 24. Properties Supported by the IBM OLE DB Provider for DB2 (continued)

Property Group	Property Set	Properties	Default Value	R/W
		DBPROP_PERSISTENTIDTYPE	DBPROPVAL_PT_NAME	R
		DBPROP_PREPAREABORTBEHAVIOR	DBPROPVAL_CB_DELETE	R
		DBPROP_PROCEDURETERM	"STORED PROCEDURE"	R
		DBPROP_PROVIDERFRIENDLYNAME	"IBM OLE DB Provider for DB2"	R
		DBPROP_PROVIDERNAME	"IBMDADB2.DLL"	R
		DBPROP_PROVIDEROLEDBVER	"02.7"	R
		DBPROP_PROVIDERVER	N/A	R
		DBPROP_QUOTEIDENTIFIERCASE	DBPROPVAL_IC_SENSITIVE	R
		DBPROP _ROWSETCONVERSIONSONCOMMAND	VARIANT_TRUE	R
		DBPROP_SCHEMATERM	"SCHEMA"	R
		DBPROP_SCHEMAUSAGE	DBPROPVAL_SU_DML_STATEMENTS | DBPROPVAL_SU_TABLE_DEFINITION | DBPROPVAL_SU_INDEX_DEFINITION | DBPROPVAL_SU_PRIVILEGE_DEFINITION	R
		DBPROP_SQLSUPPORT	DBPROPVAL_SQL_ODBC_EXTENDED | DBPROPVAL_SQL_ESCAPECLAUSES | DBPROPVAL_SQL_ANSI92_ENTRY	R
		DBPROP_SERVERNAME	N/A	R
		DBPROP_STRUCTUREDSTORAGE	DBPROPVAL_SS_ISEQUENTIALSTREAM	R
		DBPROP_SUBQUERIES	DBPROPVAL_SQ_CORRELATEDSUBQUERIES | DBPROPVAL_SQ_COMPARISON | DBPROPVAL_SQ_EXISTS | DBPROPVAL_SQ_IN | DBPROPVAL_SQ_QUANTIFIED |	R
		DBPROP_SUPPORTEDTXNDDL	DBPROPVAL_TC_ALL	R
		DBPROP_SUPPORTEDTXNISOLEVELS	DBPROPVAL_TI_CURSORSTABILITY | DBPROPVAL_TI_READCOMMITTED | DBPROPVAL_TI_READUNCOMMITTED | DBPROPVAL_TI_SERIALIZABLE |	R
		DBPROP_SUPPORTEDTXNISORETAIN	DBPROPVAL_TR_COMMIT_DC | DBPROPVAL_TR_ABORT_NO |	R
		DBPROP_TABLETERM	"TABLE"	R
		DBPROP_USERNAME	N/A	R
Initialization	DBPROPSET_DBINIT	DBPROP_AUTH_PASSWORD	N/A	R/W
		DBPROP_AUTH_PERSIST _SENSITIVE_AUTHINFO	VARIANT_FALSE	R/W
		DBPROP_AUTH_USERID	N/A	R/W
		DBPROP_INIT_DATASOURCE	N/A	R/W
		DBPROP_INIT_HWND	N/A	R/W
		DBPROP_INIT_MODE	DB_MODE_READWRITE	R/W
		DBPROP_INIT_OLEDBSERVICES	0xFFFFFFFF	R/W
		DBPROP_INIT_PROMPT	DBPROMPT_NOPROMPT	R/W
		DBPROP_INIT_PROVIDERSTRING	N/A	R/W
Rowset	DBPROPSET_ROWSET	DBPROP_ABORTPRESERVE	VARIANT_FALSE	R
		DBPROP_ACCESSORDER	DBPROPVAL_AO_RANDOM	R
		DBPROP_BLOCKINGSTORAGEOBJECTS	VARIANT_FALSE	R
		DBPROP_BOOKMARKS	VARIANT_FALSE	R/W
		DBPROP_BOOKMARKSKIPPED	VARIANT_FALSE	R
		DBPROP_BOOKMARKTYPE	DBPROPVAL_BMK_NUMERIC	R
		DBPROP_CACHEDEFERRED	VARIANT_FALSE	R/W
		DBPROP_CANFETCHBACKWARDS	VARIANT_FALSE	R/W
		DBPROP_CANHOLDROWS	VARIANT_FALSE	R
		DBPROP_CANSCROLLBACKWARDS	VARIANT_FALSE	R/W
		DBPROP_CHANGEINSERTEDROWS	VARIANT_FALSE	R
		DBPROP_COMMITPRESERVE	VARIANT_TRUE	R/W
		DBPROP_COMMANDTIMEOUT	0	R/W
		DBPROP_DEFERRED	VARIANT_FALSE	R
		DBPROP_IAccessor	VARIANT_TRUE	R
		DBPROP_IColumnsInfo	VARIANT_TRUE	R
		DBPROP_IColumnsRowset	VARIANT_TRUE	R/W
		DBPROP_IConvertType	VARIANT_TRUE	R
		DBPROP_IMultipleResults	VARIANT_FALSE	R/W

Table 24. Properties Supported by the IBM OLE DB Provider for DB2 (continued)

Property Group	Property Set	Properties	Default Value	R/W
		DBPROP_IRowset	VARIANT_TRUE	R
		DBPROP_IRowChange	VARIANT_FALSE	R/W
		DBPROP_IRowsetFind	VARIANT_FALSE	R
		DBPROP_IRowsetIdentity	VARIANT_TRUE	R
		DBPROP_IRowsetInfo	VARIANT_TRUE	R
		DBPROP_IRowsetLocate	VARIANT_FALSE	R/W
		DBPROP_IRowsetScroll	VARIANT_FALSE	R/W
		DBPROP_IRowsetUpdate	VARIANT_FALSE	R
		DBPROP_ISequentialStream	VARIANT_TRUE	R
		DBPROP_ISupportErrorInfo	VARIANT_TRUE	R
		DBPROP_LITERALBOOKMARKS	VARIANT_FALSE	R
		DBPROP_LITERALIDENTITY	VARIANT_TRUE	R
		DBPROP_LOCKMODE	DBPROPVAL_LM_SINGLEROW	R/W
		DBPROP_MAXOPENROWS	32767	R
		DBPROP_MAXROWS	0	R/W
		DBPROP_NOTIFICATIONGRANULARITY	DBPROPVAL_NT_SINGLEROW	R/W
		DBPROP_NOTIFICATION PHASES	DBPROPVAL_NP_OKTODO DBPROPBAL_NP_ABOUTTODO DBPROPVAL_NP_SYNCHAFTER DBPROPVAL_NP_FAILEDTODO DBPROPVAL_NP_DIDEVENT	R
		DBPROP_NOTIFYROWSETRELEASE	DBPROPVAL_NP_OKTODO DBPROPVAL_NP_ABOUTTODO	R
		DBPROP _NOTIFYROWSETFETCHPOSITIONCHANGE	DBPROPVAL_NP_OKTODO DBPROPVAL_NP_ABOUTTODO	R
		DBPROP_NOTIFYCOLUMNSET	DBPROPVAL_NP_OKTODO DBPROPVAL_NP_ABOUTTODO	R
		DBPROP_NOTIFYROWDELETE	DBPROPVAL_NP_OKTODO DBPROPVAL_NP_ABOUTTODO	R
		DBPROP_NOTIFYROWINSERT	DBPROPVAL_NP_OKTODO DBPROPVAL_NP_ABOUTTODO	R
		DBPROP_ORDEREDBOOKMARKS	VARIANT_FALSE	R
		DBPROP_OTHERINSERT	VARIANT_FALSE	R
		DBPROP_OTHERUPDATEDELETE	VARIANT_FALSE	R/W
		DBPROP_OWNINSERT	VARIANT_FALSE	R
		DBPROP_OWNUPDATEDELETE	VARIANT_FALSE	R
		DBPROP_QUICKRESTART	VARIANT_FALSE	R/W
		DBPROP_REMOVEDELETED	VARIANT_FALSE	R/W
		DBPROP_ROWTHREADMODEL	DBPROPVAL_RT_FREETHREAD	R
		DBPROP_SERVERCURSOR	VARIANT_TRUE	R
		DBPROP_SERVERDATAONINSERT	VARIANT_FALSE	R
		DBPROP_UNIQUEROWS	VARIANT_FALSE	R/W
		DBPROP_UPDATABILITY	0	R/W
Rowset	DBPROPSET_DB2ROWSET	DBPROP_ISLONGMINLENGTH	32000	R/W
Session	DBPROPSET_SESSION	DBPROP_SESS_AUTOCOMMITISOLEVELS	DBPROPVAL_TI_CURSORSTABILITY	R/W

Connections to Data Sources Using IBM OLE DB Provider

The following examples show how to connect to a DB2® data source using the IBM® OLE DB Provider for DB2:

Example 1: Visual Basic application using ADO:

```
Dim db As ADODB.Connection
Set db = New ADODB.Connection
db.Provider = "IBMDADB2"
db.CursorLocation = adUseClient
...
```

Example 2: C/C++ application using IDBPromptInitialize and Data Links:

```
// Create DataLinks
hr = CoCreateInstance (
     CLSID_DataLinks,
     NULL,
     CLSCTX_INPROC_SERVER,
     IID_IDBPromptInitialize,
     (void**)&pIDBPromptInitialize);

// Invoke the DataLinks UI to select the provider and data source
hr = pIDBPromptInitialize->PromptDataSource (
     NULL,
     GetDesktopWindow(),
     DBPROMPTOPTIONS_PROPERTYSHEET,
     0,
     NULL,
     NULL,
     IID_IDBInitialize,
     (IUnknown**)&pIDBInitialize);
```

Example 3: C/C++ application using IDataInitialize and Service Component:

```
hr = CoCreateInstance (
     CLSID_MSDAINITIALIZE,
     NULL,
     CLSCTX_INPROC_SERVER,
     IID_IDataInitialize,
     (void**)&pIDataInitialize);

hr = pIDataInitialize->CreateDBInstance(
     CLSID_IBMDADB2, // ClassID of IBMDADB2
     NULL,
     CLSCTX_INPROC_SERVER,
     NULL,
     IID_IDBInitialize,
     (IUnknown**)&pIDBInitialize);
```

ADO Applications

The sections that follow describe considerations for ADO applications.

ADO Connection String Keywords

To specify ADO (ActiveX Data Objects) connection string keywords, specify the keyword using the `keyword=value` format in the provider (connection) string. Delimit multiple keywords with a semicolon (;).

The following table describes the keywords supported by the IBM® OLE DB Provider for DB2®:

Table 25. Keywords Supported by the IBM OLE DB Provider for DB2

Keyword	Value	Meaning
DSN	Name of the database alias	The DB2 database alias in the database directory.
UID	User ID	The user ID used to connect to the DB2 server.
PWD	Password of UID	Password for the user ID used to connect to the DB2 server.

Other DB2 CLI configuration keywords also affect the behavior of the IBM OLE DB Provider.

Related reference:

- "CLI/ODBC configuration keywords listing by category" in the *CLI Guide and Reference, Volume 1*

Connections to Data Sources with Visual Basic ADO Applications

To connect to a DB2® data source using the IBM® OLE DB Provider for DB2, specify the IBMDADB2 provider name.

Related concepts:

- "Connections to Data Sources Using IBM OLE DB Provider" on page 232

Related tasks:

- "Building ADO applications with Visual Basic" in the *Application Development Guide: Building and Running Applications*

Updatable Scrollable Cursors in ADO Applications

The IBM® OLE DB Provider for DB2® natively supports read-only, forward-only, read-only scrollable, and updatable scrollable cursors. An ADO application that wants to access updatable scrollable cursors can set the cursor location to either `adUseClient` or `adUseServer`. Setting the cursor location to `adUseServer` causes the cursor to materialize on the server.

Limitations for ADO Applications

Following are the limitations for ADO applications:

- ADO applications calling stored procedures must have their parameters created and explicitly bound. The `Parameters.Refresh` method for automatically generating parameters is not supported for DB2 Server for VSE & VM.
- There is no support for default parameter values.
- When inserting a new row using a server-side scrollable cursor, use the AddNew() method with the Fieldlist and Values arguments. This is more efficient than calling AddNew() with no arguments following Update() calls for each column. Each AddNew() and Update() call is a separate request to the server and therefore, is less efficient than a single call to AddNew().
- Newly inserted rows are not updatable with a server-side scrollable cursor.
- Tables with long data, LOB, or Datalink columns are not updatable when using a server-side scrollable cursor.

IBM OLE DB Provider Support for ADO Methods and Properties

The IBM OLE DB Provider supports the following ADO methods and properties:

Table 26. ADO Methods and Properties Supported by the IBM OLE DB Provider for DB2

ADO	Method/Property	OLE DB Interface/Property	IBM OLE DB Support
Command Methods	Cancel	ICommand	Yes
	CreateParameter		Yes

Table 26. ADO Methods and Properties Supported by the IBM OLE DB Provider for DB2 (continued)

ADO	Method/Property	OLE DB Interface/Property	IBM OLE DB Support
	Execute		Yes
Command Properties	ActiveConnection	(ADO specific)	
	Command Text	ICommandText	Yes
	Command Timeout	ICommandProperties::SetProperties DBPROP_COMMANDTIMEOUT	Yes
	CommandType	(ADO specific)	
	Prepared	ICommandPrepare	Yes
	State	(ADO specific)	
Command Collection	Parameters	ICommandWithParameter DBSCHEMA _PROCEDURE_PARAMETERS	Yes
	Properties	ICommandProperties IDBProperties	Yes
Connection Methods	BeginTrans CommitTrans RollbackTrans	ITransactionLocal	Yes (but not nested) Yes (but not nested) Yes (but not nested)
	Execute	ICommand IOpenRowset	Yes
	Open	IDBCreateSession IDBInitialize	Yes
	OpenSchema adSchemaColumnPrivileges adSchemaColumns adSchemaForeignKeys adSchemaIndexes adSchemaPrimaryKeys adSchemaProcedureParam adSchemaProcedures adSchemaProviderType adSchemaStatistics adSchemaTablePrivileges adSchemaTables	IDBSchemaRowset	 Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes
	Cancel		Yes
Connection Properties	Attributes adXactCommitRetaining adXactRollbackRetaining	ITransactionLocal	 Yes Yes
	CommandTimeout	ICommandProperties DBPROP_COMMAND_TIMEOUT	Yes
	ConnectionString	(ADO specific)	
	ConnectionTimeout	IDBProperties DBPROP_INIT_TIMEOUT	No
	CursorLocation: adUseClient adUseNone adUseServer	 (Use OLE DB Cursor Service) (Not Used)	 Yes No Yes
	DefaultDataBase	IDBProperties DBPROP_CURRENTCATALOG	No
	IsolationLevel	ITransactionLocal DBPROP_SESS _AUTOCOMMITISOLEVELS	Yes

Table 26. ADO Methods and Properties Supported by the IBM OLE DB Provider for DB2 (continued)

ADO	Method/Property	OLE DB Interface/Property	IBM OLE DB Support
	Mode adModeRead adModeReadWrite adModeShareDenyNone adModeShareDenyRead adModeShareDenyWrite adModeShareExclusive adModeUnknown adModeWrite	IDBProperties DBPROP_INIT_MODE	 No Yes No No No No No No
	Provider	ISourceRowset::GetSourceRowset	Yes
	State	(ADO specific)	
	Version	(ADO specific)	
Connection Collection	Errors	IErrorRecords	Yes
	Properties	IDBProperties	Yes
Error Properties	Description NativeError Number Source SQLState	IErrorRecords	Yes Yes Yes Yes Yes
	HelpContext HelpFile		No No
Field Methods	AppendChunk GetChunk	ISequentialStream	Yes Yes
Field Properties	Actual Size	IAccessor IRowset	Yes
	Attributes DataFormat DefinedSize Name NumericScale Precision Type	IColumnInfo	 Yes Yes Yes Yes Yes Yes
	OriginalValue	IRowsetUpdate	Yes (Cursor Service)
	UnderlyingValue	IRowsetRefresh IRowsetResynch	Yes (Cursor Service) Yes (Cursor Service)
	Value	IAccessor IRowset	Yes
Field Collection	Properties	IDBProperties IRowsetInfo	Yes
Parameter Methods	AppendChunk	ISequentialStream	Yes
	Attributes Direction Name NumericScale Precision Scale Size Type	ICommandWithParameter DBSCHEMA _PROCEDURE_PARAMETERS	 Yes No Yes Yes Yes Yes Yes
	Value	IAccessor ICommand	Yes
Parameter Collection	Properties		Yes

Table 26. ADO Methods and Properties Supported by the IBM OLE DB Provider for DB2 (continued)

ADO	Method/Property	OLE DB Interface/Property	IBM OLE DB Support
RecordSet Methods	AddNew	IRowsetChange	Yes
	Cancel		Yes
	CancelBatch	IRowsetUpdate::Undo	Yes (Cursor Service)
	CancelUpdate		Yes (Cursor Service)
	Clone	IRowsetLocate	Yes
	Close	IAccessor IRowset	Yes
	CompareBookmarks		No
	Delete	IRowsetChange	Yes
	GetRows	IAccessor IRowset	Yes
	Move	IRowset IRowsetLocate	Yes
	MoveFirst	IRowset IRowsetLocate	Yes
	MoveNext	IRowset IRowsetLocate	Yes
	MoveLast	IRowsetLocate	Yes
	MovePrevious	IRowsetLocate	Yes
	NextRecordSet	IMultipleResults	Yes
	Open	ICommand IOpenRowset	Yes
	Requery	ICommand IOpenRowset	Yes
	Resync	IRowsetRefresh	Yes (Cursor Service)
	Supports	IRowsetInfo	Yes
	Update UpdateBatch	IRowsetChange IRowsetUpdate	Yes Yes (Cursor Service)
RecordSet Properties	AbsolutePage	IRowsetLocate IRowsetScroll	Yes Yes[1]
	AbsolutePosition	IRowsetLocate IRowsetScroll	Yes Yes[1]
	ActiveConnection	IDBCreateSession IDBInitialize	Yes
	BOF	(ADO specific)	
	Bookmark	IAccessor IRowsetLocate	Yes
	CacheSize	cRows in IRowsetLocate IRowset	Yes
	CursorType adOpenDynamic adOpenForwardOnly adOpenKeySet adOpenStatic	ICommandProperties	 No Yes Yes Yes
	EditMode	IRowsetUpdate	Yes (Cursor Service)
	EOF	(ADO specific)	

Table 26. ADO Methods and Properties Supported by the IBM OLE DB Provider for DB2 (continued)

ADO	Method/Property	OLE DB Interface/Property	IBM OLE DB Support
	Filter	IRowsetLocate IRowsetView IRowsetUpdate IViewChapter IViewFilter	No
	LockType	ICommandProperties	Yes
	MarshallOption		No
	MaxRecords	ICommandProperties IOpenRowset	Yes
	PageCount	IRowsetScroll	Yes[1]
	PageSize	(ADO specific)	
	Sort	(ADO specific)	
	Source	(ADO specific)	
	State	(ADO specific)	
	Status	IRowsetUpdate	Yes (Cursor Service)
Notes: 1. The values to be returned are approximations. Deleted rows will not be skipped.			
RecordSet Collection	Fields	IColumnInfo	Yes
	Properties	IDBProperties IRowsetInfo::GetProperties	Yes

C and C++ Applications

The sections that follow describe considerations for C and C++ applications.

Compilation and Linking of C/C++ Applications and the IBM OLE DB Provider

C/C++ applications that use the constant CLSID_IBMDADB2 must include the `ibmdadb2.h` file, which can be found in the `SQLLIB\include` directory. These applications must define the DBINITCONSTANTS before the include statement. The following example shows the correct sequence of statements:

```
#define DBINITCONSTANTS
#include "ibmdadb2.h"
```

Connections to Data Sources in C/C++ Applications using the IBM OLE DB Provider

To connect to a DB2® data source using the IBM® OLE DB Provider for DB2 in a C/C++ application, you can use one of the two OLE DB core interfaces, `IDBPromptInitialize` or `IDataInitialize`, or you can call the COM API CoCreateInstance. The `IDataInitialize` interface is exposed by the OLE DB Service Component, and the `IDBPromptInitialize` is exposed by the Data Links Component.

Related concepts:

• "Connections to Data Sources Using IBM OLE DB Provider" on page 232

Related tasks:

- "Building ADO applications with Visual C++" in the *Application Development Guide: Building and Running Applications*

MTS and COM+ Distributed Transactions

The sections that follow describe considerations for MTS and COM+ distributed transactions.

MTS and COM+ Distributed Transaction Support and the IBM OLE DB Provider

OLE DB applications running in either a Microsoft® Transaction Server (MTS) environment on Windows® NT or a Component Services (COM+) environment on Windows 2000 can use the `ITransactionJoin` interface to participate in distributed transactions with multiple DB2® Universal Database, host, and iSeries database servers as well as other resource managers that comply with the MTS/COM+ specifications.

Prerequisites:

To use the MTS or COM+ distributed transaction support offered by the IBM® OLE DB Provider for DB2, ensure that your server meets the following prerequisites.

Note: These requirements are only for the Windows machine where the DB2 client is installed.

- Windows NT® with MTS at Version 2.0 with Microsoft Hotfix 0772 or later

  MTS Version 2.0 for Windows NT is available as part of the Windows NT 4.0 Option Pack. You can download the Option Pack from:

  `http://www.microsoft.com/ntserver/nts/downloads/recommended/NT4OptPk/`

- Windows 2000 with Service Pack 3 or later

Related concepts:

- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager" on page 646
- "Loosely coupled support with Microsoft Component Services (COM+)" on page 648

Enablement of MTS Support in DB2 Universal Database for C/C++ Applications

To run a C or C++ application in MTS or COM+ transactional mode, you can create the IBMDADB2 data source instance using the `DataLink` interface. You could also use `CoCreateInstance`, get a session object, and use `JoinTransaction`. See the description of how to connect a C or C++ application to a data source for more information.

To run an ADO application in MTS or COM+ transactional mode, see the description of how to connect a C or C++ application to a data source.

To use a component in an MTS or COM+ package in transactional mode, set the Transactions property of the component to one of the following values:

- "Required"

- "Required New"
- "Supported"

For information about these values, see the MTS documentation.

Related concepts:

- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager" on page 646
- "Loosely coupled support with Microsoft Component Services (COM+)" on page 648

Chapter 12. OLE DB .NET Data Provider

OLE DB .NET Data Provider

The OLE DB .NET Data Provider uses the IBM® DB2® OLE DB Driver, which is referred to in a `ConnectionString` object as `IBMDADB2`. The connection string keywords supported by the OLE DB .NET Data Provider are the same as those supported by the IBM OLE DB Provider for DB2. Also, the OLE DB .NET Data Provider has the same restrictions as the IBM DB2 OLE DB Provider. There are additional restrictions for the OLE DB .NET Data Provider, which are identified in the topic: OLE DB .NET Data Provider restrictions.

In order to use the OLE DB .NET Data Provider, you must have the .NET Framework Version 1.1 installed.

For DB2 Universal Database™ for AS/400® and iSeries™, the following fix is required on the server: APAR ii13348.

The following are all the supported connection keywords for the OLE DB .NET Data Provider:

Table 27. **ConnectionString** *keywords for the OLE DB .NET Data Provider*

Keyword	Value	Meaning
`PROVIDER`	`IBMDADB2`	Specifies the IBM OLE DB Provider for DB2 (required)
`DSN` or `Data Source`	database alias	The DB2 database alias as cataloged in the database directory
`UID`	user ID	The user ID used to connect to the DB2 server
`PWD`	password	The password for the user ID used to connect to the DB2 server

The following is an example of creating an `OleDbConnection` to connect to the SAMPLE database:

```
[Visual Basic .NET]
Dim con As New OleDbConnection("Provider=IBMDADB2;" +
    "Data Source=sample;UID=userid;PWD=password;")
con.Open()

[C#]
OleDbConnection con = new OleDbConnection("Provider=IBMDADB2;" +
                "Data Source=sample;UID=userid;PWD=password;" );
con.Open()
```

OLE DB .NET Data Provider restrictions

The following table identifies usage restrictions for the IBM OLE DB .NET Data Provider:

Table 28. IBM OLE DB .NET Data Provider restrictions

Class or feature	Restriction description	DB2 servers affected
ASCII character streams	You cannot use ASCII character streams with `OleDbParameters` when using `DbType.AnsiString` or `DbType.AnsiStringFixedLength`. The OLE DB .NET Data Provider will throw the following exception: `"Specified cast is not valid"` **Workaround:** Use `DbType.Binary` instead of using `DbType.AnsiString` or `DbType.AnsiStringFixedLength`.	All
`ADORecord`	`ADORecord` is not supported.	All

`ADORecordSet` and `Timestamp`	As documented in MSDN, the `ADORecordSet` variant time resolves to one second. Consequently, all fractional seconds are lost when a DB2 `Timestamp` column is stored into a `ADORecordSet`. Similarly, after filling a `DataSet` from a `ADORecordSet`, the `Timestamp` columns in the `DataSet` will not have any fractional seconds.
	
	Workaround:
	
	This workaround only works for DB2 Universal Database for Linux, UNIX, and Windows, Version 8.1, FixPak 4 or later. In order to avoid the loss of fraction of seconds, you can set the following CLI keyword:
	`MAPTIMESTAMPDESCRIBE = 2`
	
	This keyword will describe the `Timestamp` as a WCHAR(26). To set the keyword, execute the following command from a DB2 Command Window:
	`db2 update cli cfg for section common using MAPTIMESTAMPDESCRIBE 2`
		All

`Chapters`	Chapters are not supported.	All
Key information	The OLE DB .NET Data Provider cannot retrieve key information when opening an `IDataReader` at the same time.	DB2 for VM/VSE
Key information from stored procedures	The OLE DB .NET Data Provider can retrieve key information about a result set returned by a stored procedure only from DB2 Universal Database for Linux, UNIX, and Windows. This is because the DB2 servers for platforms other than Linux, UNIX, and Windows do not return extended describe information for the result sets opened in the stored procedure. In order to retrieve key information of a result set returned by a stored procedure on DB2 Universal Database for Linux, UNIX, and Windows, you need to set the following registry variable on the DB2 server: `db2set DB2_APM_PERFORMANCE=8` Setting this server-side DB2 registry variable will keep the result set meta-data available on the server for a longer period of time, thus allowing OLE DB to successfully retrieve the key information. However, depending on the server workload, the meta-data might not be available long enough before the OLE DB Provider queries for the information. As such, there is no guarantee that the key information will always be available for result sets returned from a store procedure. In order to retrieve any key information about a CALL statement, the application must execute the CALL statement. Calling `OleDbDataAdapter.FillSchema()` or `OleDbCommand.ExecuteReader(CommandBehavior.SchemaOnly | CommandBehavior.KeyInfo)`, will not actually execute the stored procedure call. Therefore, you will not retrieve any key information for the result set that is to be returned by the stored procedure.	All

Table 28. IBM OLE DB .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

Key information from batched SQL statements	When using batched SQL statements that return multiple results, the `FillSchema()` method attempts to retrieve schema information only for the first SQL statement in the batched SQL statement list. If this statement does not return a result set then no table is created. For example:
	`[C#]`
	`cmd.CommandText = "INSERT INTO ORG(C1) VALUES(1000); SELECT C1 FROM ORG;";`
	`da = new OleDbDataAdapter(cmd);`
	`da.FillSchema(ds, SchemaType.Source);`
	
	No table will be created in the data set because the first statement in the batch SQL statement is an ″INSERT″ statement, which does not return a result set.
		All

`OleDbCommandBuilder`	The UPDATE, DELETE and INSERT statements automatically generated by the `OleDbCommandBuilder` are incorrect if the SELECT statement contains any columns of the following data types: • CLOB • BLOB • DBCLOB • LONG VARCHAR • LONG VARCHAR FOR BIT DATA • LONG VARGRAPHIC If you are connecting to a DB2 server other than DB2 Universal Database for Linux, Unix and Windows, then columns of the following data types also cause this problem: • VARCHAR[1] • VARCHAR FOR BIT DATA[1] • VARGRAPHIC[1] • REAL • FLOAT or DOUBLE • TIMESTAMP **Notes:** 1. Columns of these data types are applicable if they are defined to be VARCHAR values greater than 254 bytes, VARCHAR values FOR BIT DATA greater than 254 bytes, or VARGRAPHICs greater than 127 bytes. This condition is only valid if you are connecting to a DB2 server other than DB2 Universal Database for Linux, Unix and Windows. The `OleDbCommandBuilder` generates SQL statements that use all of the selected columns in an equality comparison in the WHERE clause, but the data types listed previously cannot be used in an equality comparison. **Note:** Note that this restriction will affect the `IDbDataAdapter.Update()` method that relies on the `OleDbCommandBuilder` to automatically generate the UPDATE, DELETE, and INSERT statements. The UPDATE operation will fail if the generated statement contains any one of the data types listed previously. **Workaround:** You will need to explicitly remove all columns that are of the data types listed previously from the WHERE clause of the generated SQL statement. It is recommended that you code your own UPDATE, DELETE and INSERT statements.	All
`OleDbCommandBuilder.DeriveParameters`	Case-sensitivity is important when using `DeriveParameters()`. The stored procedure name specified in the `OleDbCommand.CommandText` needs to be in the same case as how it is stored in the DB2 system catalog tables. To see how stored procedure names are stored, call OpenSchema(OleDbSchemaGuid.Procedures) without supplying the procedure name restriction. This will return all the stored procedure names. By default, DB2 stores stored procedure names in uppercase, so most often, you need to specify the stored procedure name in uppercase.	All

Table 28. IBM OLE DB .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected
`OleDbCommandBuilder.DeriveParameters`	The `OleDbCommandBuilder.DeriveParameters()` method does not include the `ReturnValue` parameter in the generated `OleDbParameterCollection`. SqlClient and the DB2 .NET Data Provider by default adds the parameter with `ParameterDirection.ReturnValue` to the generated `ParameterCollection`.	All
`OleDbCommandBuilder.DeriveParameters`	The `OleDbCommandBuilder.DeriveParameters()` method will fail for overloaded stored procedures. If you have multiple stored procedures of the name "MYPROC" with each of them taking a different number of parameters or different type of parameter, the `OleDbCommandBuilder.DeriveParameters()` will retrieve all the parameters for all the overloaded stored procedures.	All
`OleDbCommandBuilder.DeriveParameters`	If the application does not qualify a stored procedure with a schema, `DeriveParameters()` will return all the parameters for that procedure name. Therefore, if multiple schemas exist for the same procedure name, `DeriveParameters()` will return all the parameters for all the procedures with the same name.	All
`OleDbConnection.ChangeDatabase`	The `OleDbConnection.ChangeDatabase()` method is not supported.	All
`OleDbConnection.ConnectionString`	Use of nonprintable characters such as '\b', '\a' or '\O' in the connection string will cause an exception to be thrown. The following keywords have restrictions: **`Data Source`** The data source is the name of the database, not the server. You can specify the SERVER keyword, but it is ignored by the IBMDADB2 provider. **`Initial Catalog and Connect Timeout`** These keywords are not supported. In general, the OLE DB .NET Data Provider will ignore all unrecognized and unsupported keywords. However, specifying these keywords will cause the following exception: `Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No work was done.` **`ConnectionTimeout`** `ConnectionTimeout` is read only.	All

`OleDbConnection.GetOleDbSchemaTable`	Restriction values are case-sensitive, and need to match the case of the database objects stored in the system catalog tables, which defaults to uppercase.
	
	For instance, if you have created a table in the following manner:
	`CREATE TABLE abc(c1 SMALLINT)`
	
	DB2 will store the table name in uppercase ("ABC") in the system catalog. Therefore, you will need to use "ABC" as the restriction value. For instance:
	`schemaTable = con.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, new object[] { null, null, "ABC", "TABLE" });`
	
	Workaround:
	
	If you need case-sensitivity or spaces in your data definitions, you must put quotation marks around them. For example:
	`cmd.CommandText = "create table \"Case Sensitive\"(c1 int)";`
	`cmd.ExecuteNonQuery();`
	`tablename = "\"Case Sensitive\"";`
	`schemaTable = con.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, new object[] { null, null, tablename, "TABLE" });`
		All

`OleDbDataAdapter` and `DataColumnMapping`	The source column name is case-sensitive. It needs to match the case as stored in the DB2 catalogs, which by default is uppercase. For example: `colMap = new DataColumnMapping("EMPNO", "Employee ID");`	All

Table 28. IBM OLE DB .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected
`OleDbDataReader.GetSchemaTable`	The OLE DB .NET Data Provider is not able to retrieve extended describe information from servers that do not return extended describe information. if you are connecting to a server that does not support extended describe (the affected servers), the following columns in the metadata table returned from `IDataReader.GetSchemaTable()` are invalid: • `IsReadOnly` • `IsUnique` • `IsAutoIncrement` • `BaseSchemaName` • `BaseCatalogName`	DB2 for OS/390, version 7 or lower DB2 for OS/400 DB2 for VM/VSE
Stored procedures: no column names for result sets	The DB2 for OS/390 version 6.1 server does not return column names for result sets returned from a stored procedure. The OLE DB .NET Data Provider maps these unnamed columns to their ordinal position (for example, ″1″, ″2″ ″3″). This is contrary to the mapping documented in MSDN: `"Column1", "Column2", "Column3"`.	DB2 for OS/390 version 6.1

Connection pooling in OLE DB .NET Data Provider applications

The OLE DB .NET Data Provider automatically pools connections using OLE DB session pooling. Connection string arguments can be used to enable or disable OLE DB services including pooling. For example, the following connection string will disable OLE DB session pooling and automatic transaction enlistment.

```
Provider=IBMDADB2;OLE DB Services=-4;Data Source=SAMPLE;
```

The following table describes the ADO connection string attributes you can use to set the OLE DB services:

Table 29. Setting OLE DB services by using ADO connection string attributes

Services enabled	Value in connection string
All services (the default)	″OLE DB Services = -1;″
All services except pooling	″OLE DB Services = -2;″
All services except pooling and auto-enlistment	″OLE DB Services = -4;″
All services except client cursor	″OLE DB Services = -5;″
All services except client cursor and pooling	″OLE DB Services = -6;″
No services	″OLE DB Services = 0;″

For more information about OLE DB session pooling or resource pooling, as well as how to disable pooling by overriding OLE DB provider service defaults, see the OLE DB Programmer's Reference in the MSDN library located at http://msdn.microsoft.com/library.

Time columns in OLE DB .NET Data Provider applications

The following sections describe how to implement time columns in OLE DB .NET Data Provider applications.

Inserting using parameter markers:

You want to insert a time value into a Time column:

```
command.CommandText = "insert into mytable(c1) values( ? )";
```

where column c1 is a Time column. Here are two methods to bind a time value to the parameter marker:

Using `OleDbParameter.OleDbType = OleDbType.DBTime`

Because OleDbType.DBTime maps to a TimeSpan object, you must supply a TimeSpan object as the parameter value. The parameter value cannot be a String or a DateTime object, it must be a TimeSpan object. For example:

```
p1.OleDbType = OleDbType.DBTime;
p1.Value = TimeSpan.Parse("0.11:20:30");
rowsAffected = cmd.ExecuteNonQuery();
```

The format of the TimeSpan is represented as a string in the format ″[-]d.hh:mm:ss.ff″ as documented in the MSDN documentation.

Using `OleDbParameter.OleDbType = OleDbType.DateTime`

This will force the OLE DB .NET Data Provider to convert the parameter value to a DateTime object, instead of a TimeSpan object, consequently the parameter value can be any valid string/object that can be converted into a DateTime object. This means values such as ″11:20:30″ will work. The value can also be a DateTime object. The value cannot be a TimeSpan object since a TimeSpan object cannot be converted to a DateTime object -- TimeSpan doesn't implement IConvertible.

For example:

```
p1.OleDbType = OleDbType.DBTimeStamp;
p1.Value = "11:20:30";
rowsAffected = cmd.ExecuteNonQuery();
```

Retrieval:

To retrieve a time column you need to use the `IDataRecord.GetValue()` method or the `OleDbDataReader.GetTimeSpan()` method.

For example:

```
TimeSpan ts1 = ((OleDbDataReader)reader).GetTimeSpan( 0 );
TimeSpan ts2 = (TimeSpan) reader.GetValue( 0 );
```

ADORecordset objects in OLE DB .NET Data Provider applications

Following are considerations regarding the use of `ADORecordset` objects.

- The ADO type `adDBTime` class is mapped to the .NET Framework `DateTime` class. `OleDbType.DBTime` corresponds to a `TimeSpan` object.
- You cannot assign a `TimeSpan` object to an `ADORecordset` object's `Time` field. This is because the `ADORecordset` object's `Time` field expects a `DateTime` object. When you assign a `TimeSpan` object to an `ADORecordsetobject, you will get the following message:`

  ```
  Method's type signature is not Interop compatible.
  ```

  You can only populate the `Time` field with a `DateTime` object, or a `String` that can be parsed into a `DateTime` object.
- When you fill a `DataSet` with a `ADORecordset` using the `OleDbDataAdapter`, the `Time` field in the `ADORecordset` is converted to a `TimeSpan` column in the `DataSet`.

- `Recordsets` do not store primary keys or constraints. Therefore, no key information is added when filling out a `DataSet` from a `Recordset` using the `MissingSchemaAction.AddWithKey`.

Chapter 13. ODBC .NET Data Provider

ODBC .NET Data Provider

The ODBC .NET Data Provider makes ODBC calls to a DB2® data source using the DB2 CLI Driver. Therefore, the connection string keywords supported by the ODBC .NET Data Provider are the same as those supported by the DB2 CLI driver. Also, the ODBC .NET Data Provider has the same restrictions as the DB2 CLI driver. There are additional restrictions for the ODBC .NET Data Provider, which are identified in the topic: ODBC .NET Data Provider restrictions.

In order to use the ODBC .NET Data Provider, you must have the .NET Framework Version 1.1 installed. For DB2 Universal Database for AS/400® and iSeries™, the following fix is required on the server: APAR II13348.

The following are the supported connection keywords for the ODBC .NET Data Provider:

Table 30. **ConnectionString** *keywords for the ODBC .NET Data Provider*

Keyword	Value	Meaning
`DSN`	database alias	The DB2 database alias as cataloged in the database directory
`UID`	user ID	The user ID used to connect to the DB2 server
`PWD`	password	The password for the user ID used to connect to the DB2 server

The following is an example of creating an `OdbcConnection` to connect to the SAMPLE database:

```
[Visual Basic .NET]
Dim con As New OdbcConnection("DSN=sample;UID=userid;PWD=password;")
con.Open()

[C#]
OdbcConnection con = new OdbcConnection("DSN=sample;UID=userid;PWD=password;");
con.Open()
```

ODBC .NET Data Provider restrictions

The following table identifies usage restrictions for the IBM ODBC .NET Data Provider:

Table 31. IBM ODBC .NET Data Provider restrictions

Class or feature	Restriction description	DB2 servers affected
ASCII character streams	You cannot use ASCII character streams with `OdbcParameters` when using `DbType.AnsiString` or `DbType.AnsiStringFixedLength`. The ODBC .NET Data Provider will throw the following exception: `"Specified cast is not valid"` **Workaround:** Use `DbType.Binary` instead of using `DbType.AnsiString` or `DbType.AnsiStringFixedLength`.	All

`Command.Prepare`	Before executing a command (`Command.ExecuteNonQuery` or `Command.ExecuteReader`), you must explicitly run `OdbcCommand.Prepare()` if the `CommandText` has changed since the last prepare. If you do not call `OdbcCommand.Prepare()` again, the ODBC .NET Data Provider will execute the previously prepared `CommandText`.
	
	For Example:
	`[C#]`
	`command.CommandText="select CLOB('ABC') from table1";`
	`command.Prepare();`
	`command.ExecuteReader();`
	`command.CommandText="select CLOB('XYZ') from table2";`
	`command.ExecuteReader();  // This ends up re-executing the first statement`
		All

CommandBehavior. SequentialAccess	When using `IDataReader.GetChars()` to read from a reader created with `CommandBehavior.SequentialAccess`, you must allocate a buffer that is large enough to hold the entire column. Otherwise, you will hit the following exception:
	`Requested range extends past the end of the array.`
	`  at System.Runtime.InteropServices.Marshal.Copy(Int32 source,`
	`  Char[] destination, Int32 startIndex, Int32 length)`
	`  at System.Data.Odbc.OdbcDataReader.GetChars(Int32 i,`
	`  Int64 dataIndex, Char[] buffer, Int32 bufferIndex, Int32 length)`
	`  at OleRestrict.TestGetCharsAndBufferSize(IDbConnection con)`
	
	The following example demonstrates how to allocate an adequate buffer:
	`CREATE TABLE myTable(c0 int, c1 CLOB(10K))`
	`SELECT c1 FROM myTable;`
	
	`[C#]`
	`cmd.CommandText = "SELECT c1 from myTable";`
	`IDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess);`
	
	`Int32 iChunkSize = 10;`
	`Int32 iBufferSize = 10;`
	`Int32 iFieldOffset = 0;`
	
	`Char[] buffer = new Char[ iBufferSize ];`
	
	`reader.Read();`
	`reader.GetChars(0, iFieldOffset, buffer, 0, iChunkSize);`
	
	The call to `GetChars()` will throw the following exception:
	`"Requested range extends past the end of the array"`
	
	To ensure that `GetChars()` does not throw the above exception, you must set the `BufferSize` to the size of the column, as follows:
	`Int32 iBufferSize = 10000;`
	
	Note that the value of 10,000 for `iBufferSize` corresponds to the value of 10K allocated to the CLOB column `c1`.
		All

Table 31. IBM ODBC .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

CommandBehavior. SequentialAccess	The ODBC .NET Data Provider throws the following exception when there is no more data to read when using `OdbcDataReader.GetChars()`:
	`NO_DATA - no error information available`
	`at System.Data.Odbc.OdbcConnection.HandleError(HandleRef hrHandle, SQL_HANDLE hType, RETCODE retcode)`
	`at System.Data.Odbc.OdbcDataReader.GetData(Int32 i, SQL_C sqlctype, Int32 cb)`
	`at System.Data.Odbc.OdbcDataReader.GetChars(Int32 i, Int64 dataIndex, Char[] buffer, Int32 bufferIndex, Int32 length)`
		All

CommandBehavior. SequentialAccess	You may not be able to use large chunksizes, such as a value of 5000, when using `OdbcDataReader.GetChars()`. When you attempt to use a large chunk size, the ODBC .NET Data Provider will throw the following exception:
	`Object reference not set to an instance of an object.`
	`at System.Runtime.InteropServices.Marshal.Copy(Int32 source, Char[] destination, Int32 startIndex, Int32 length)`
	`at System.Data.Odbc.OdbcDataReader.GetChars(Int32 i, Int64 dataIndex, Char[] buffer, Int32 bufferIndex, Int32 length)`
	`at OleRestrict.TestGetCharsAndBufferSize(IDbConnection con)`
		All

Connection pooling	The ODBC .NET Data Provider does not control connection pooling. Connection pooling is handled by the ODBC Driver Manager. For more information on connection pooling, see the ODBC Programmer's Reference in the MSDN library located at http://msdn.microsoft.com/library.	All
DataColumnMapping	The case of the source column name needs to match the case used in the system catalog tables, which is upper-case by default.	All

Decimal columns	Parameter markers are not supported for Decimal columns.
	
	You generally use `OdbcType.Decimal` for an `OdbcParameter` if the target `SQLType` is a `Decimal` column; however, when the ODBC .NET Data Provider sees the `OdbcType.Decimal, it binds the parameter using C-type of SQL_C_WCHAR` and `SQLType of SQL_VARCHAR`, which is invalid.
	
	For example:
	`[C#]`
	`cmd.CommandText = "SELECT dec_col FROM MYTABLE WHERE dec_col > ? ";`
	`OdbcParameter p1 = cmd.CreateParameter();`
	`p1.DbType = DbType.Decimal;`
	`p1.Value = 10.0;`
	`cmd.Parameters.Add(p1);`
	`IDataReader rdr = cmd.ExecuteReader();`
	
	You will get an exception:
	`ERROR [07006] [IBM][CLI Driver][SQLDS/VM] SQL0301N The value of input host variable or parameter number "" cannot be used because of its data type. SQLSTATE=07006`
	
	Workaround:
	
	Instead of using `OdbcParameter` values, use literals exclusively.
		DB2 for VM/VSE

Table 31. IBM ODBC .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

Key information	The schema name used to qualify the table name (for example, MYSCHEMA.MYTABLE) must match the connection user ID. The ODBC .NET Data Provider is unable to retrieve any key information in which the specified schema is different from the connection user id.
	
	For example:
	`CREATE TABLE USERID2.TABLE1(c1 INT NOT NULL PRIMARY KEY);`
	
	`[C#]`
	`// Connect as user bob`
	`odbcCon = new OdbcConnection("DSN=sample;UID=bob;PWD=mypassword");`
	
	`OdbcCommand cmd = odbcCon.CreateCommand();`
	
	`// Select from table with schema USERID2`
	`cmd.CommandText="SELECT * FROM USERID2.TABLE1";`
	
	`// Fails - No key info retrieved`
	`da.FillSchema(ds, SchemaType.Source);`
	
	`// Fails - SchemaTable has no primary key`
	`cmd.ExecuteReader(CommandBehavior.KeyInfo)`
	
	`// Throws exception because no primary key`
	`cbuilder.GetUpdateCommand();`
		All

Key information	The ODBC .NET Data Provider cannot retrieve key information when opening a IDataReader at the same time. When the ODBC .NET Data Provider opens a IDataReader, a cursor on the server is opened. If key information is requested, it will then call SQLPrimaryKeys() or SQLStatistic() to get the key information, but these schema functions will open another cursor. Since DB2 for VM/VSE does not support cursor withhold, the first cursor is then closed. Consequently, IDataReader.Read() calls to the IDataReader will result in the following exception:
	`System.Data.Odbc.OdbcException: ERROR [HY010] [IBM][CLI Driver]`
	`  CLI0125E  Function sequence error. SQLSTATE=HY010`
	
	Workaround:
	
	You will need to retrieve key information first then retrieve the data.
	
	For example:
	`[C#]`
	`OdbcCommand cmd = odbcCon.CreateCommand();`
	`OdbcDataAdapter da = new OdbcDataAdapter(cmd);`
	
	`cmd.CommandText  = "SELECT * FROM MYTABLE";`
	
	`// Use FillSchema to retrieve just the schema information`
	`da.FillSchema(ds, SchemaType.Source);`
	`// Use FillSchema to retrieve just the schema information`
	`da.Fill(ds);`
		DB2 for VM/VSE

Key information	You must refer to database objects in your SQL statements using the same case that the database objects are stored in the system catalog tables. By default database objects are stored in uppercase in the system catalog tables, so most often, you need to use uppercase. The ODBC .NET Data Provider scans SQL statements to retrieve database object names and passes them to schema functions such as SQLPrimaryKeys and SQLStatistics, which issue queries for these objects in the system catalog tables. The database object references must match exactly how they are stored in the system catalog tables, otherwise, an empty result set is returned.	DB2 for OS/390 DB2 for OS/400 DB2 for VM/VSE
Key information for batched non-select SQL statements	The ODBC .NET Data Provider is unable to retrieve any key information for a batch statement that does not start with ″SELECT″.	DB2 for OS/390 DB2 for OS/400 DB2 for VM/VSE

Table 31. IBM ODBC .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

LOB columns	The ODBC .NET Data Provider does not support LOB datatypes. Consequently, whenever the DB2 server returns a SQL_CLOB (-99), SQL_BLOB (-98) or SQL_DBCLOB (-350) the ODBC .NET Data Provider will throw the following exception:
	`"Unknown SQL type - -98"  (for Blob column)`
	`"Unknown SQL type - -99"  (for Clob column)`
	`"Unknown SQL type - -350"  (for DbClob column)`
	Any methods that directly or indirectly access LOB columns will fail.
	Workaround:
	Set the CLI/ODBC `LongDataCompat` keyword to 1. Doing so will force the DB2 CLI driver to make the following data type mappings to data types the ODBC .NET Data Provider will understand:
	• SQL_CLOB to SQL_LONGVARCHAR
	• SQL_BLOB to SQL_LONGVARBINARY
	• SQL_DBCLOB to SQL_WLONGVARCHAR
	To set the `LongDataCompat` keyword, run the following DB2 command from a DB2 command window on the client machine:
	`db2 update cli cfg for section common using longdatacompat 1`
	You can also set this keyword in your application, using the connection string as follows:
	`[C#]`
	`OdbcConnection con =`
	`  new OdbcConnection("DSN=SAMPLE;UID=uid;PWD=mypwd;LONGDATACOMPAT=1;");`
	For a list of all the CLI/ODBC keywords, see the following topic: UID CLI/ODBC configuration keyword in the DB2 Universal Database CLI Guide and Reference.
		All

`OdbcCommand.Cancel`	Executing statements after running `OdbcCommand.Cancel` can lead to the following exception: `"ERROR [24000] [Microsoft][ODBC Driver Manager] Invalid cursor state"`	All

`OdbcCommandBuilder`	Case-sensitivity is important when using the `OdbcCommandBuilder` to automatically generate UPDATE, DELETE, and INSERT statements. By default, DB2 stores schema information (such as table names, and column names) in the system catalog tables in upper case, unless they have been explicitly created with case-sensitivity (by adding quotes around database objects during create-time). As such, your SQL statements must match the case that is stored in the catalogs (which by default is uppercase).
	For example, if you created a table using the following statement:
	`"db2 create table mytable (c1 int) "`
	then DB2 will store the table name ″mytable″ in the system catalog tables as ″MYTABLE″.
	The following code example demonstrates proper use the `OdbcCommandBuilder`class:
	`[C#]`
	`OdbcCommand cmd = odbcCon.CreateCommand();`
	`cmd.CommandText  = "SELECT * FROM MYTABLE";`
	`OdbcDataAdapter da = new OdbcDataAdapter(cmd);`
	`OdbcCommandBuilder cb = new OdbcCommandBuilder(da);`
	`OdbcCommand updateCmd = cb.GetUpdateCommand();`
	In this example, if you do not refer to the table name in upper-case characters, then you will get the following exception:
	`"Dynamic SQL generation for the UpdateCommand is not`
	`supported against a SelectCommand that does not return`
	`any key column information."`
		All

Table 31. IBM ODBC .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

OdbcCommandBuilder	The commands generated by the `OdbcCommandBuilder` are incorrect when the SELECT statement contains the following column data types:
	`REAL`
	`FLOAT or DOUBLE`
	`TIMESTAMP`
	
	These data types cannot be used in the WHERE clause for SELECT statements.
		DB2 for OS/390
		DB2 for OS/400
		DB2 for VM/VSE

OdbcCommandBuilder. DeriveParameters	The `DeriveParameters()` method is mapped to `SQLProcedureColumns` and it uses the `CommandText` property for the name of the stored procedure. Since `CommandText` does not contain the name of the stored procedure (using full ODBC call syntax), `SQLProcedureColumns` is called with the procedure name identified according to the ODBC call syntax. For example: `"{ CALL myProc(?) }"` This which will result in an empty result set, where no columns are found for the procedure).	All
OdbcCommandBuilder. DeriveParameters	To use `DeriveParameters()`, specify the stored procedure name in the `CommandText` (for example, `cmd.CommandText = "MYPROC"`). The procedure name must match the case stored in the system catalog tables. `DeriveParameters()` will return all the parameters for that procedure name it finds in the system catalog tables. Remember to change the `CommandText` back to the full ODBC call syntax before executing the statement.	All
OdbcCommandBuilder. DeriveParameters	The `ReturnValue` parameter is not returned for the ODBC .NET Data Provider.	All
OdbcCommandBuilder. DeriveParameters	`DeriveParameters()` does not support fully qualified stored procedure names. For example, calling `DeriveParameters()` for `CommandText = "MYSCHEMA.MYPROC"` will fail. Here, no parameters are returned.	All
OdbcCommandBuilder. DeriveParameters	`DeriveParameters()` will not work for overloaded stored procedures. The `SQLProcedureColumns` will return all the parameters for all versions of the stored procedure.	All
`OdbcConnection.` `ChangeDatabase`	The `OdbcConnection.ChangeDatabase()` method is not supported.	All
`OdbcConnection.` `ConnectionString`	• The `Server` keyword is ignored. • The `Connect Timeout` keyword is ignored. DB2 CLI does not support connection timeouts, so setting this property will not affect the driver. • Connection pooling keywords are ignored. Specifically, this affects the following keywords: `Pooling`, `Min Pool Size`, `Max Pool Size`, `Connection Lifetime` and `Connection Reset`.	All
OdbcDataReader. GetSchemaTable	The ODBC .NET Data Provider is not able to retrieve extended describe information from servers that do not return extended describe information. Therefore, if you are connecting to a server that does not support extended describe (the affected servers), the following columns in the metadata table returned from `IDataReader.GetSchemaTable()` are invalid: • `IsReadOnly` • `IsUnique` • `IsAutoIncrement` • `BaseSchemaName` • `BaseCatalogName`	DB2 for OS/390, version 7 or lower DB2 for OS/400 DB2 for VM/VSE

Table 31. IBM ODBC .NET Data Provider restrictions (continued)

Class or feature	Restriction description	DB2 servers affected

Stored procedures	To call a stored procedure, you need to specify the full ODBC call syntax.
	
	For example, to call the stored procedure, `MYPROC`, that takes a VARCHAR(10) as a parameter:
	`[C#]`
	`OdbcCommand cmd = odbcCon.CreateCommand();`
	`cmd.CommandType = CommandType.Text;`
	`cmd.CommandText = "{ CALL MYPROC(?) }"`
	`OdbcParameter p1 = cmd.CreateParameter();`
	`p1.Value = "Joe";`
	`p1.OdbcType = OdbcType.NVarChar;`
	`cmd.Parameters.Add(p1);`
	`cmd.ExecuteNonQuery();`
	
	Note: Note that you must use the full ODBC call syntax even if you are using `CommandType.StoredProcedure`. This is documented in MSDN, under the OdbcCommand.CommandText Property.
		All

Stored procedures: no column names for result sets	The DB2 for OS/390 version 6.1 server does not return column names for result sets returned from a stored procedure. The ODBC .NET Data Provider maps these unnamed columns to their ordinal position (for example, ″1″, ″2″ ″3″). This is contrary to the mapping documented in MSDN: `"Column1", "Column2", "Column3"`.	DB2 for OS/390 version 6.1
Unique index promotion to primary key	The ODBC .NET Data Provider promotes nullable unique indexes to primary keys. This is contrary to the MSDN documentation, which states that nullable unique indexes should not be promoted to primary keys.	All

Part 4. Java

Chapter 14. Introduction to Java application support

DB2® Universal Database provides driver support for client applications and applets that are written in Java™ using JDBC, and for embedded SQL for Java (SQLJ).

JDBC is an application programming interface (API) that Java applications use to access relational databases. DB2 Universal Database™ support for JDBC lets you write Java applications that access local DB2 data or remote relational data on a server that supports DRDA®.

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM®, Oracle®, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.

In general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL. However, because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

This topic discusses the Java application development environment provided by DB2 Universal Database.

According to the JDBC specification, there are four types of JDBC driver architectures:

Type 1
Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a type 1 driver.

Type 2
Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited.

Type 3
Drivers that use a pure Java client and communicate with a server using a database-independent protocol. The server then communicates the client's requests to the data source.

Type 4
Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

DB2 Version 8 supports a type 2 driver and a driver that combines type 2 and type 4 JDBC implementations. DB2 Version 8 also supports a type 3 driver, although this driver is deprecated. The JDBC drivers in previous releases of DB2 UDB for Linux, UNIX® and Windows® were built on DB2 CLI (Call Level Interface). The DB2 Version 8 type 2 and type 3 drivers continue to use the DB2 CLI interface to communicate with DB2 UDB servers. DB2 Version 8 adds a new DB2 Universal JDBC Driver that is written completely in Java. The drivers that are supported in DB2 Version 8 are:

DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC type 2 driver) (deprecated as of DB2 V8.2):

The DB2 JDBC type 2 driver lets Java applications make calls to DB2 through JDBC. Calls to the DB2 JDBC type 2 driver are translated to Java native methods. The Java applications that use this driver must run on a DB2 client, through which JDBC requests flow to the DB2 server. DB2 Connect™ Version 8 must be installed before the DB2 JDBC application driver can be used to access DB2 UDB for iSeries™ data sources or data sources in the DB2 for OS/390 or z/OS environments.

The DB2 JDBC type 2 driver supports these JDBC and SQLJ functions:

- Most of the methods that are described in the JDBC 1.2 specification, and some of the methods that are described in the JDBC 2.0 specification. See Comparison of driver support for JDBC APIs.
- SQLJ statements that perform equivalent functions to all JDBC methods
- Connection pooling
- Distributed transactions
- Java user-defined functions and stored procedures

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows will not be supported in future releases of DB2. You should therefore consider moving to the DB2 Universal JDBC Driver.

DB2 JDBC Type 3 Driver for Linux, UNIX and Windows (deprecated as of DB2 V8.1):

The DB2 JDBC type 3 driver, also known as the applet or net driver, consists of a JDBC client and a JDBC server. The DB2 JDBC applet driver can be loaded by a Web browser along with the applet, or the applet driver can be used in standalone Java applications. When the applet requests a connection to a DB2 database server, the applet driver opens a TCP/IP socket to the DB2 JDBC applet server on the machine where the Web server is running. After a connection is set up, the applet driver sends each of the subsequent database access requests from the applet to the JDBC server through the TCP/IP connection. The JDBC server then makes corresponding DB2 calls to perform the task. On completion, the JDBC server sends the results back to the JDBC client through the connection. The JDBC server process is db2jd.

The DB2 JDBC Type 3 Driver for Linux, UNIX and Windows will not be supported in future releases of DB2. You should therefore consider moving to the DB2 Universal JDBC Driver.

DB2 Universal JDBC driver (type 2 and type 4):

The DB2 Universal JDBC Driver is a single driver that includes JDBC type 2 and JDBC type 4 behavior, as well as SQLJ support. When an application loads the DB2 Universal JDBC Driver, a single driver instance is loaded for type 2 and type 4 implementations. The application can make type 2 and type 4 connections using this single driver instance. The type 2 and type 4 connections can be made concurrently. DB2 Universal JDBC Driver type 2 driver behavior is referred to as *DB2 Universal JDBC Driver type 2 connectivity.* DB2 Universal JDBC Driver type 4 driver behavior is referred to as *DB2 Universal JDBC Driver type 4 connectivity.*

The DB2 Universal JDBC Driver is an entirely new driver, rather than a follow-on to any other DB2 JDBC drivers. Therefore, you can expect some differences in behavior between this driver and other drivers.

The DB2 Universal JDBC Driver supports these JDBC and SQLJ functions:

- Most of the methods that are described in the JDBC 1.2 and JDBC 2.0 specifications, and some of the methods that are described in the JDBC 3.0 specifications. See Comparison of driver support for JDBC APIs.
- SQLJ statements that perform equivalent functions to all JDBC methods.
- Connections that are enabled for connection pooling. WebSphere Application Server or another application server does the connection pooling.
- Implementation of Java user-defined functions and stored procedures (Universal Type 2 Connectivity only).
- Global transactions that run under WebSphere® Application Server Version 5.0 and above.
- Support for distributed transaction management. This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS) and Java Transaction API (JTA) specifications, which conform to the X/Open standard for global transactions (*Distributed Transaction Processing: The XA Specification*, available from www.opengroup.org) .

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "SQLJ differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 432
- "Comparison of driver support for JDBC APIs" on page 376

Chapter 15. JDBC application programming

The sections that follow contain information about writing JDBC applications.

Basic JDBC application programming concepts

The topics that follow contain basic information about writing JDBC applications.

Basic steps in writing a JDBC application

Writing a JDBC application has much in common with writing an SQL application in any other language: In general, you need to do the following things:

- Access the Java™ packages that contain JDBC methods.
- Declare variables for sending data to or retrieving data from DB2® tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks is somewhat different.

Figure 4 on page 264 is a simple program that demonstrates each task. This program runs on the DB2 Universal JDBC Driver.

```
import java.sql.*;                                                          1

public class EzJava
{
  public static void main(String[] args)
  {
    String urlPrefix = "jdbc:db2:";
    String url;
    String empNo;                                                           2
    Connection con;
    Statement stmt;
    ResultSet rs;

    System.out.println ("**** Enter class EzJava");

    // Check the that first argument has the correct form for the portion
    // of the URL that follows jdbc:db2:, as described
    // in the Connecting to a data source using the DriverManager
    // interface with the DB2 Universal JDBC Driver topic.
    // For example, for Universal Driver type 2 connectivity,
    // args[0] might be MVS1DB2M. For Universal
    // Driver type 4 connectivity, args[0] might
    // be //stlmvs1:10110/MVS1DB2M.
    if (args.length==0)
    {
      System.err.println ("Invalid value. First argument appended to "+
       "jdbc:db2: must specify a valid URL.");
      System.exit(1);
    }
    url = urlPrefix + args[0];

    try
    {
      // Load the DB2 Universal JDBC Driver
      Class.forName("com.ibm.db2.jcc.DB2Driver");                           3a
      System.out.println("**** Loaded the JDBC driver");

      // Create the connection using the DB2 Universal JDBC Driver
      con = DriverManager.getConnection (url);                              3b
      // Commit changes manually
      con.setAutoCommit(false);
      System.out.println("**** Created a JDBC connection to the data source");

      // Create the Statement
      stmt = con.createStatement();                                         4a
      System.out.println("**** Created JDBC Statement object");

      // Execute a query and generate a ResultSet instance
      rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");                 4b
      System.out.println("**** Creaed JDBC ResultSet object");

      // Print all of the employee numbers to standard output device
      while (rs.next()) {
        empNo = rs.getString(1);
        System.out.println("Employee number = " + empNo);
      }
      System.out.println("**** Fetched all rows from JDBC ResultSet");
```

Figure 4. Simple JDBC application (Part 1 of 2)

```
      // Close the ResultSet
      rs.close();
      System.out.println("**** Closed JDBC ResultSet");

      // Close the Statement
      stmt.close();
      System.out.println("**** Closed JDBC Statement");

      // Connection must be on a unit-of-work boundary to allow close
      con.commit();
      System.out.println ( "**** Transaction committed" );

      // Close the connection
      con.close();                                                          6
      System.out.println("**** Disconnected from data source");

      System.out.println("**** JDBC Exit from class EzJava - no errors");

    }

    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.out.println("Exception: " + e);
      e.printStackTrace();
    }

    catch(SQLException ex)                                                  5
    {
      System.err.println("SQLException information");
      while(ex!=null) {
        System.err.println ("Error msg: " + ex.getMessage());
        System.err.println ("SQLSTATE: " + ex.getSQLState());
        System.err.println ("Error code: " + ex.getErrorCode());
        ex.printStackTrace();
        ex = ex.getNextException(); // For drivers that support chained exceptions
      }
    }
  }  // End main
}    // End EzJava
```

Figure 4. Simple JDBC application (Part 2 of 2)

Notes to Figure 4 on page 264:

1	This statement imports the java.sql package, which contains the JDBC core API. For information on other Java packages that you might need to access, see Access Java packages for JDBC support.
2	`String` variable empNo performs the function of a host variable. That is, it is used to hold data retrieved from an SQL query. See Declare variables in JDBC applications for more information.
3a and **3b**	These two sets of statements demonstrate how to connect to a data source using one of two available interfaces. See Connect to a data source using JDBC for more details.
4a and **4b**	These two sets of statements demonstrate how to perform a SELECT in JDBC. For information on how to perform other SQL operations, see Execute SQL in a JDBC application.
5	This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see Handle an SQLException under the DB2 Universal JDBC Driver. For information on handling SQL warnings, see Handle SQL warnings in a JDBC application.
6	This statement disconnects the application from the data source. See Close the connection to the data source.

Related concepts:

- "Java packages for JDBC support" on page 266

- "Variables in JDBC applications" on page 266
- "JDBC interfaces for executing SQL" on page 276
- "How JDBC applications connect to a data source" on page 267

Related tasks:

- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Handling an SQLWarning under the DB2 Universal JDBC Driver" on page 287

Java packages for JDBC support

Before you can invoke JDBC methods, you need to be able to access all or parts of various Java™ packages that contain those methods. You can do that either by importing the packages or specific classes, or by using the fully-qualified class names. You might need the following packages or classes for your JDBC program:

java.sql
Contains the core JDBC API.

javax.naming
Contains classes and interfaces for Java Naming and Directory Interface (JNDI), which is often used for implementing a `DataSource`.

javax.sql
Contains JDBC 2.0 standard extensions.

javax.transaction
Contains JDBC support for distributed transactions for the DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver).

com.ibm.db2.jcc
Contains the DB2-specific implementation of JDBC for the DB2 Universal JDBC driver.

COM.ibm.db2.jdbc
Contains the DB2-specific implementation of the JDBC for the DB2 JDBC Type 2 Driver.

Variables in JDBC applications

As in any other Java™ application, when you write JDBC applications, you declare variables. In Java applications, those variables are known as Java identifiers. Some of those identifiers have the same function as host variables in other languages: they hold data that you pass to or retrieve from DB2® tables. Identifier `empNo` in the sample program in Basic steps in writing a JDBC application is an example of a Java `String` identifier that holds data that you retrieve from a CHAR column of a DB2 table.

Your choice of Java data types can affect performance because DB2 picks better access paths when the data types of your Java variables map closely to the DB2 data types. Java, JDBC, and SQL data types shows the recommended mappings of Java data types and JDBC data types to SQL data types.

Related concepts:

- "Basic steps in writing a JDBC application" on page 263

Related reference:

- "Java, JDBC, and SQL data types" on page 365

How JDBC applications connect to a data source

Before you can execute SQL statements in any SQL program, you must connect to a database server. In JDBC, a database server is known as a *data source*.

Figure 5 shows how a Java™ application connects to a data source for a type 2 driver or DB2 Universal JDBC Driver type 2 connectivity.

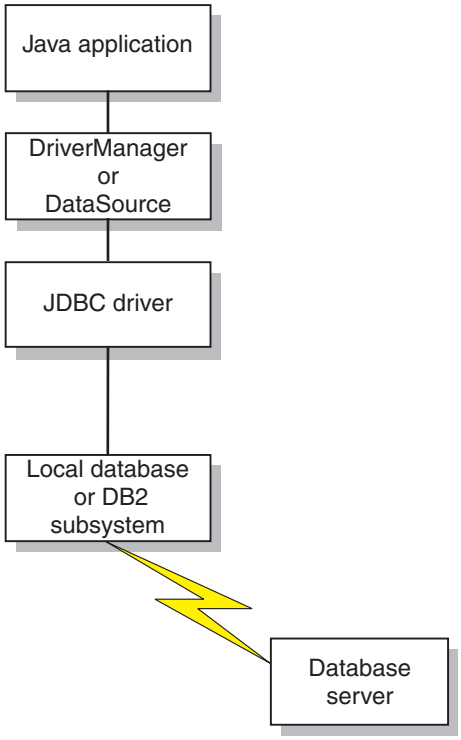
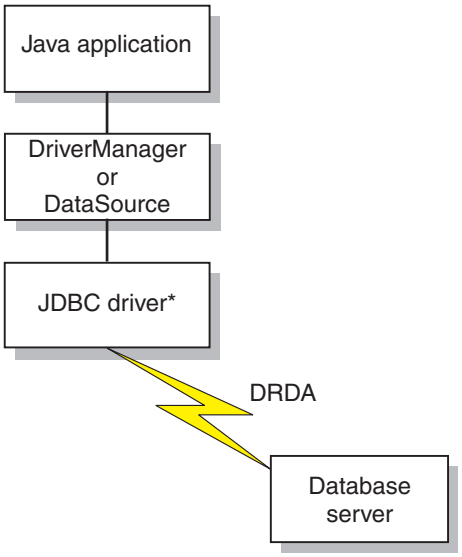


Figure 5. Java application flow for a type 2 driver or DB2 Universal JDBC Driver type 2 connectivity

Figure 6 on page 268 shows how a Java application connects to a data source for DB2 Universal JDBC Driver type 4 connectivity.



*Java byte code executed under JVM

Figure 6. Java application flow for DB2 Universal JDBC Driver type 4 connectivity

The way that you connect to a data source depends on the version of JDBC that you use. Connecting using the `DriverManager` interface is available for all levels of JDBC. Connecting using the `DataSource` interface is available with JDBC 2.0 and above.

Related concepts:

- "How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver" on page 268

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver

A JDBC application can establish a connection to a data source using the JDBC `DriverManager` interface, which is part of the `java.sql` package.

The Java™ application first loads the JDBC driver by invoking the `Class.forName` method. After the application loads the driver, it connects to a database server by invoking the `DriverManager.getConnection` method.

For the DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver), you load the driver by invoking the `Class.forName` method with the following argument:

```
COM.ibm.db2.jdbc.app.DB2Driver
```

The following code demonstrates loading the DB2 JDBC Type 2 Driver:

```
try {
  // Load the DB2 JDBC Type 2 Driver with DriverManager
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

The `catch` block is used to print an error if the driver is not found.

After you load the driver, you connect to the data source by invoking the `DriverManager.getConnection` method. You can use one of the following forms of `getConnection`:

```
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

The *url* argument represents a data source.

For the DB2 JDBC Type 2 Driver, specify a URL of the following form:

Syntax for a URL for the DB2 JDBC Type 2 Driver:

The parts of the URL have the following meanings:

jdbc:db2:
jdbc:db2: indicates that the connection is to a DB2 UDB server.

database
A database alias. The alias refers to the DB2 database catalog entry on the DB2 client.

The *info* argument is an object of type `java.util.Properties` that contains a set of driver properties for the connection. Specifying the *info* argument is an alternative to specifying *property=value* strings in the URL.

Specifying a user ID and password for a connection: There are several ways to specify a user ID and password for a connection:

- Use the form of the `getConnection` method that specifies *user* and *password*.
- Use the form of the `getConnection` method that specifies *info*, after setting the user and password properties in a `java.util.Properties` object.

Example: Setting the user ID and password in user and password parameters:

```
String url = "jdbc:db2:toronto";
                                        // Set URL for data source
String user = "db2adm";
String password = "db2adm";
Connection con = DriverManager.getConnection(url, user, password);
                                        // Create connection
```

Example: Setting the user ID and password in a `java.util.Properties` object:

```
Properties properties = new Properties(); // Create Properties object
properties.put("user", "db2adm");         // Set user ID for connection
properties.put("password", "db2adm");     // Set password for connection
String url = "jdbc:db2:toronto";
```

```
                                        // Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create connection
```

Related concepts:

- "Security under the DB2 JDBC Type 2 Driver" on page 443

Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver

A JDBC application can establish a connection to a data source using the JDBC `DriverManager` interface, which is part of the `java.sql` package.

The Java™ application first loads the JDBC driver by invoking the `Class.forName` method. After the application loads the driver, it connects to a database server by invoking the `DriverManager.getConnection` method.

For the DB2 Universal JDBC Driver, you load the driver by invoking the `Class.forName` method with the following argument:

```
com.ibm.db2.jcc.DB2Driver
```

For compatibility with previous JDBC drivers, you can use the following argument instead:

```
COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
```

The following code demonstrates loading the DB2 Universal JDBC Driver:

```
try {
  // Load the DB2® Universal JDBC Driver with DriverManager
  Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch (ClassNotFoundException e) {
     e.printStackTrace();
}
```

The `catch` block is used to print an error if the driver is not found.

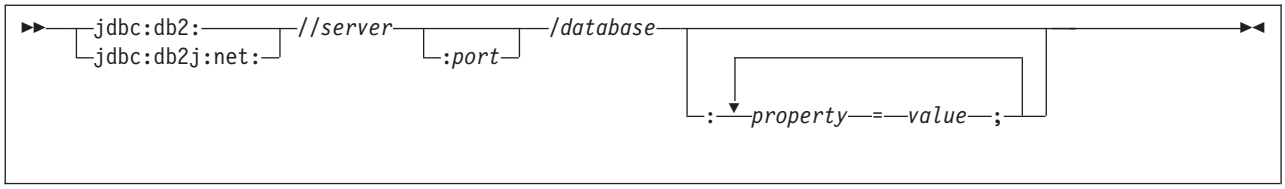
After you load the driver, you connect to the data source by invoking the `DriverManager.getConnection` method. You can use one of the following forms of `getConnection`:

```
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

The *url* argument represents a data source, and indicates what type of JDBC connectivity you are using.

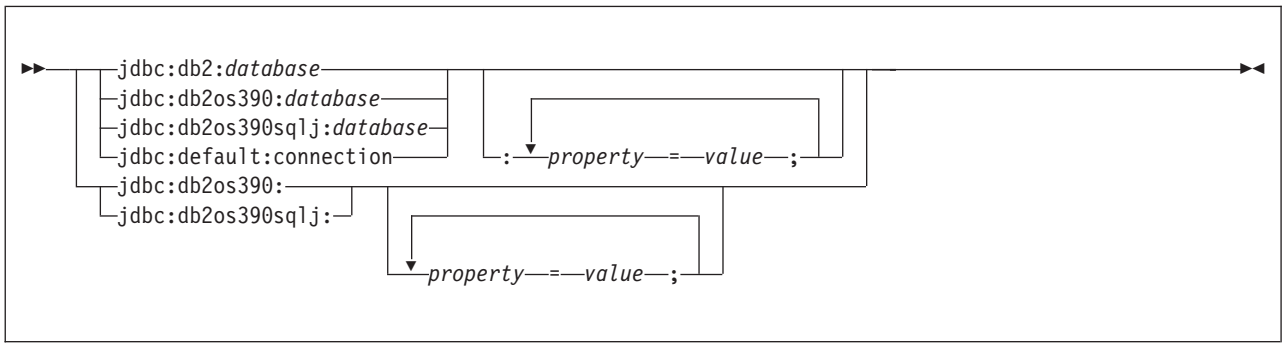
For DB2 Universal JDBC Driver type 4 connectivity, specify a URL of the following form:

Syntax for a URL for Universal Type 4 Connectivity:



For DB2 Universal JDBC Driver type 2 connectivity, specify a URL of one of the following forms:

Syntax for a URL for Universal Type 2 Connectivity:



The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j:net:
The meanings of the initial portion of the URL are:

jdbc:db2:
Indicates that the connection is to a server in the DB2 UDB family.

jdbc:db2j:net:
Indicates that the connection is to a remote IBM® Cloudscape™ server.

server
The domain name or IP address of the database server.

port
The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

database
A name for the database server. This name depends on whether Universal Type 4 Connectivity or Universal Type 2 Connectivity is used.

For Universal Type 4 Connectivity:

- If the connection is to a DB2 for z/OS server, *database* is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

  ```
  SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
  ```

- If the connection is to a DB2 UDB for Linux, UNIX and Windows server, *database* is the database name that is defined during installation.
- If the connection is to an IBM Cloudscape server, the *database* is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

  ```
  "c:/databases/testdb"
  ```

For Universal Type 2 Connectivity:

- *database* is the database name that is defined during installation, if the value of the serverName connection property is null. If the value of serverName property is not null, *database* is a database alias.

property=value;
A property for the JDBC connection. For the definitions of these properties, see Properties for the DB2 Universal JDBC Driver.

The *info* argument is an object of type `java.util.Properties` that contains a set of driver properties for the connection. Specifying the *info* argument is an alternative to specifying *property=value* strings in the URL. See Properties for the DB2 Universal JDBC Driver for the properties that you can specify.

Specifying a user ID and password for a connection: There are several ways to specify a user ID and password for a connection:

- Use the form of the `getConnection` method that specifies *url* with `property=value;` clauses, and include the user and password properties in the URL.
- Use the form of the `getConnection` method that specifies *user* and *password*.
- Use the form of the `getConnection` method that specifies *info*, after setting the user and password properties in a `java.util.Properties` object.

Example: Setting the user ID and password in a URL:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose:" +
  "user=db2adm;password=db2adm;";
                                        // Set URL for data source
Connection con = DriverManager.getConnection(url);
                                        // Create connection
```

Example: Setting the user ID and password in user and password parameters:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
                                        // Set URL for data source
String user = "db2adm";
String password = "db2adm";
Connection con = DriverManager.getConnection(url, user, password);
                                        // Create connection
```

Example: Setting the user ID and password in a `java.util.Properties` object:

```
Properties properties = new Properties(); // Create Properties object
properties.put("user", "db2adm");         // Set user ID for connection
properties.put("password", "db2adm");     // Set password for connection
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
                                        // Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create connection
```

Related concepts:

- "Security under the DB2 Universal JDBC Driver" on page 444

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Connecting to a data source using the DataSource interface

Using `DriverManager` to connect to a data source reduces portability because the application must identify a specific JDBC driver class name and driver URL. The

driver class name and driver URL are specific to a JDBC vendor, driver implementation, and data source. If your applications need to be portable among data sources, you should use the DataSource interface.

When you connect to a data source using the DataSource interface, you use a DataSource object. It is possible to create and use the DataSource object in the same application, as you do with the DriverManager interface. Figure 7 shows an example for the DB2 Universal JDBC Driver:

Figure 7. Creating and using a DataSource object in the same application

```
import java.sql.*;        // JDBC base
import javax.sql.*;       // JDBC 2.0 standard extension APIs
import com.ibm.db2.jcc.*; // DB2® Universal JDBC Driver   1
                          // interfaces
DB2SimpleDataSource db2ds=new DB2SimpleDataSource();      2
db2ds.setDatabaseName("db2loc1");                         3
                          // Assign the location name
db2ds.setDescription("Our Sample Database");
                          // Description for documentation
db2ds.setUser("john");
                          // Assign the user ID
db2ds.setPassword("db2");
                          // Assign the password
Connection con=db2ds.getConnection();                     4
                          // Create a Connection object
```

1. Import the package that contains the implementation of the DataSource interface.
2. Creates a DB2DataSource object. DB2DataSource is one of the DB2 implementations of the DataSource interface. See Create and deploy DataSource objects for information on DB2's DataSource implementations.
3. The setDatabaseName, setDescription, setUser, and setPassword methods assign attributes to the DB2DataSource object. See Properties for the DB2 Universal JDBC Driver for information about the attributes that you can set for a DB2DataSource object under the DB2 Universal JDBC Driver.
4. Establishes a connection to the data source that DB2DataSource object db2ds represents.

However, a more flexible way to use a DataSource object is for your system administrator to create and manage it separately, using WebSphere® or some other tool. The program that creates and manages a DataSource object also uses the Java™ Naming and Directory Interface (JNDI) to assign a logical name to the DataSource object. The JDBC application that uses the DataSource object can then refer to the object by its logical name, and does not need any information about the underlying data source. In addition, your system administrator can modify the data source attributes, and you do not need to change your application program.

To learn more about using WebSphere to deploy DataSource objects, go to this URL on the Web:

`http://www.ibm.com/software/webservers/appserv/`

To learn about deploying DataSource objects yourself, see Create and deploy DataSource objects.

You can use the DataSource interface and the DriverManager interface in the same application, but for maximum portability, it is recommended that you use only the DataSource interface to obtain connections.

The remainder of this topic explains how to create a connection using a DataSource object, given that the system administrator has already created the object and assigned a logical name to it.

To obtain a connection using a DataSource object, you need to follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Create a Context object to use in the next step. The Context interface is part of the Java Naming and Directory Interface (JNDI), not JDBC.
3. In your application program, use JNDI to get the DataSource object that is associated with the logical data source name.
4. Use the DataSource.getConnection method to obtain the connection.

   You can use one of the following forms of the getConnection method:

   ```
   getConnection();
   getConnection(String user, String password);
   ```

   Use the second form if you need to specify a user ID and password for the connection that are different from the ones that were specified when the DataSource was deployed.

Figure 8 shows an example of the code that you need in your application program to obtain a connection using a DataSource object, given that the logical name of the data source that you need to connect to is jdbc/sampledb. The numbers to the right of selected statements correspond to the previously-described steps.

Figure 8. Obtaining a connection using a DataSource object

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
Context ctx=new InitialContext();                          2
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");     3
Connection con=ds.getConnection();                         4
```

Related tasks:

- "Creating and deploying DataSource objects" on page 311

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Setting the isolation level for a JDBC transaction

To set the isolation level for a unit of work within a JDBC program, use the Connection.setTransactionIsolation(int *level*) method. Table 32 shows the values of *level* that you can specify in the Connection.setTransactionIsolation method and their DB2® equivalents.

Table 32. Equivalent JDBC and DB2 isolation levels

JDBC value	DB2 isolation level
TRANSACTION_SERIALIZABLE	Repeatable read
TRANSACTION_REPEATABLE_READ	Read stability
TRANSACTION_READ_COMMITTED	Cursor stability

Table 32. Equivalent JDBC and DB2 isolation levels (continued)

JDBC value	DB2 isolation level
TRANSACTION_READ_UNCOMMITTED	Uncommitted read

You can change the isolation level only at the beginning of a transaction.

Related concepts:

- "JDBC connection objects" on page 275

JDBC connection objects

When you connect to a data source by either connection method, you create a `Connection` object, which represents the connection to the data source. You use this `Connection` object to do the following things:

- Create `Statement`, `PreparedStatement`, and `CallableStatement` objects for executing SQL statements. These are discussed in Execute SQL in a JDBC application.
- Gather information about the data source to which you are connected. This process is discussed in Use DatabaseMetaData to learn about a data source.
- Commit or roll back transactions. You can commit transactions manually or automatically. These operations are discussed in Commit or roll back a JDBC transaction.
- Close the connection to the data source. This operation is discussed in Close a connection to a JDBC data source.

Related concepts:

- "JDBC interfaces for executing SQL" on page 276

Related tasks:

- "Closing a connection to a JDBC data source" on page 276
- "Committing or rolling back JDBC transactions" on page 275
- "Using DatabaseMetaData to learn about a data source" on page 301

Committing or rolling back JDBC transactions

In JDBC, to commit or roll back transactions explicitly, use the `commit` or `rollback` methods. For example:

```
Connection con;
...
con.commit();
```

If autocommit mode is on, DB2® performs a commit operation after every SQL statement completes. To determine whether autocommit mode is on, invoke the `Connection.getAutoCommit` method. To set autocommit mode on, invoke the `Connection.setAutoCommit(true)` method. To set autocommit mode off, invoke the `Connection.setAutoCommit(false)` method.

Related concepts:

- "Savepoints in JDBC applications" on page 294

Related tasks:

- "Making batch updates in JDBC applications" on page 304
- "Closing a connection to a JDBC data source" on page 276

Closing a connection to a JDBC data source

When you have finished with a connection to a data source, it is *essential* that you close the connection to the data source. Doing this releases the `Connection` object's DB2® and JDBC resources immediately. To close the connection to the data source, use the `close` method. For example:

```
Connection con;
...
con.close();
```

If autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.

Related concepts:

- "How JDBC applications connect to a data source" on page 267

JDBC interfaces for executing SQL

You execute SQL statements in a traditional SQL program to insert, update, and delete data in tables, retrieve data from the tables, or call stored procedures. To perform the same functions in a JDBC program, you invoke methods that are defined in the following interfaces:

- The `Statement` interface supports all SQL statement execution. The following interfaces inherit methods from the `Statement` interface:
  - The `PreparedStatement` interface supports any SQL statement containing input parameter markers. Parameter markers represent input variables. The `PreparedStatement` interface can also be used for SQL statements with no parameter markers.

    With the DB2 Universal JDBC Driver, the `PreparedStatement` interface can be used to call stored procedures that have input parameters and no output parameters, and that return no result sets.
  - The `CallableStatement` interface supports the invocation of a stored procedure.

    The `CallableStatement` interface can be used to call stored procedures with input parameters, output parameters, or input and output parameters, or no parameters. With the DB2 Universal JDBC Driver, you can also use the `Statement` interface to call stored procedures, but those stored procedures must have no parameters.
- The `ResultSet` interface provides access to the results that a query generates. The `ResultSet` interface has the same purpose as the cursor that is used in SQL applications in other languages.

For a complete list of DB2® support for JDBC interfaces, see Comparison of driver support for JDBC APIs.

Related tasks:

- "Using the PreparedStatement.executeQuery method to retrieve data from DB2" on page 280
- "Using the PreparedStatement.executeUpdate method to update data in DB2 tables" on page 279

- "Using the Statement.executeQuery method to retrieve data from DB2 tables" on page 277
- "Using the Statement.executeUpdate method to create and modify DB2 objects" on page 277

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Using the Statement.executeUpdate method to create and modify DB2 objects

You can use the `Statement.executeUpdate` method to do the following things:

- Execute data definition statements, such as CREATE, ALTER, DROP, GRANT, REVOKE
- Execute INSERT, UPDATE and DELETE statements that do not contain parameter markers
- With the DB2 Universal JDBC Driver, execute the CALL statement to call stored procedures that have no parameters and that return no result sets.

To execute these SQL statements, you need to perform these steps:

1. Invoke the `Connection.createStatement` method to create a `Statement` object.
2. Invoke the `Statement.executeUpdate` method to perform the SQL operation.
3. Invoke the `Statement.close` method to close the `Statement` object.

For example, suppose that you want to execute this SQL statement:

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

The following code creates `Statement` object `stmt`, executes the UPDATE statement, and returns the number of rows that were updated in `numUpd`. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement();               // Create a Statement object  1
numUpd = stmt.executeUpdate(
  "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");             2
                                            // Perform the update
stmt.close();                               // Close Statement object     3
```

Figure 9. Using Statement.executeUpdate

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Comparison of driver support for JDBC APIs" on page 376

Using the Statement.executeQuery method to retrieve data from DB2 tables

To retrieve data from a table using a SELECT statement with no parameter markers, you can use the `Statement.executeQuery` method. This method returns a

result table in a ResultSet object. After you obtain the result table, you need to use ResultSet methods to move through the result table and obtain the individual column values from each row.

With the DB2 Universal JDBC Driver, you can also use the Statement.executeQuery method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set. If the stored procedure returns multiple result sets, you need to use the Statement.execute method. See Retrieve multiple result sets from a stored procedure in a JDBC application for more information.

This topic discusses the simplest kind of ResultSet, which is a read-only ResultSet in which you can only move forward, one row at a time. The DB2 Universal JDBC Driver also supports updatable and scrollable ResultSets. These are discussed in Specify updatability, scrollability, and holdability for ResultSets in JDBC applications.

To retrieve rows from a table using a SELECT statement with no parameter markers, you need to perform these steps:

1. Invoke the Connection.createStatement method to create a Statement object.
2. Invoke the Statement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
3. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods. XXX represents a data type. See Comparison of driver support for JDBC APIs for a list of supported getXXX and setXXX methods.
4. Invoke the ResultSet.close method to close the ResultSet object.
5. Invoke the Statement.close method to close the Statement object when you have finished using that object.

For example, the following code demonstrates how to retrieve all rows from the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
String empNo;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();     // Create a Statement object          1
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");                   2
                                  // Get the result table from the query
while (rs.next()) {               // Position the cursor                3
 empNo = rs.getString(1);             // Retrieve only the first column value
 System.out.println("Employee number = " + empNo);
                                  // Print the column value
}
rs.close();                       // Close the ResultSet                4
stmt.close();                     // Close the Statement                5
```

Figure 10. Using Statement.executeQuery

Related tasks:

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 297
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 309

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Using the PreparedStatement.executeUpdate method to update data in DB2 tables

The `Statement.executeUpdate` method works if you update DB2® tables with constant values. However, updates often need to involve passing values in variables to DB2 tables. To do that, you use the `PreparedStatement.executeUpdate` method.

With the DB2 Universal JDBC Driver, you can also use `PreparedStatement.executeUpdate` to call stored procedures that have input parameters and no output parameters, and that return no result sets.

When you execute an SQL statement many times, you can get better performance by creating the SQL statement as a `PreparedStatement`.

For example, the following UPDATE statement lets you update the employee table for only one phone number and one employee number:

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

Suppose that you want to generalize the operation to update the employee table for any set of phone numbers and employee numbers. You need to replace the constant phone number and employee number with variables:

```
UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?
```

Variables of this form are called parameter markers. To execute an SQL statement with parameter markers, you need to perform these steps:

1. Invoke the `Connection.prepareStatement` method to create a `PreparedStatement` object.
2. Invoke the `PreparedStatement.setXXX` methods to pass values to the variables.
3. Invoke the `PreparedStatement.executeUpdate` method to update the table with the variable values.
4. Invoke the `PreparedStatement.close` method to close the `PreparedStatement` object when you have finished using that object.

The following code performs the previous steps to update the phone number to '4657' for the employee with employee number '000010'. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                 // Create a PreparedStatement object  1
pstmt.setString(1,"4657");       // Assign value to first parameter    2
pstmt.setString(2,"000010");     // Assign value to second parameter
numUpd = pstmt.executeUpdate();  // Perform the update                 3
pstmt.close();                   // Close the PreparedStatement object 4
```

Figure 11. Using PreparedStatement.executeUpdate for an SQL statement with parameter markers

You can also use the PreparedStatement.executeUpdate method for statements that have no parameter markers. The steps for executing a PreparedStatement object with no parameter markers are similar to executing a PreparedStatement object with parameter markers, except you skip step 2. The following example demonstrates these steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
                                // Create a PreparedStatement object  1
numUpd = pstmt.executeUpdate();   // Perform the update               3
pstmt.close();                    // Close the PreparedStatement object  4
```

Figure 12. Using PreparedStatement.executeUpdate for an SQL statement without parameter markers

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Comparison of driver support for JDBC APIs" on page 376

Using the PreparedStatement.executeQuery method to retrieve data from DB2

To retrieve data from a table using a SELECT statement with parameter markers, you use the PreparedStatement.executeQuery method. This method returns a result table in a ResultSet object. After you obtain the result table, you need to use ResultSet methods to move through the result table and obtain the individual column values from each row.

With the DB2 Universal JDBC Driver, you can also use the PreparedStatement.executeQuery method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set and has only input parameters. If the stored procedure returns multiple result sets, you need to use the Statement.execute method. See Retrieve multiple result sets from a stored procedure in a JDBC application for more information.

To retrieve rows from a table using a SELECT statement with parameter markers, you need to perform these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.
2. Invoke PreparedStatement.setXXX methods to pass values to the input parameters.
3. Invoke the PreparedStatement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
4. In a loop, position the cursor using the ResultSet.next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
5. Invoke the ResultSet.close method to close the ResultSet object.
6. Invoke the PreparedStatement.close method to close the PreparedStatement object when you have finished using that object.

For example, the following code demonstrates how to retrieve rows from the employee table for a specific employee. The numbers to the right of selected statements correspond to the previously-described steps.

```
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement(
  "SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
                                  // Create a PreparedStatement object   1
pstmt.setString(1,"000010");      // Assign value to input parameter     2

rs = pstmt.executeQuery();        // Get the result table from the query 3
while (rs.next()) {               // Position the cursor                 4
 empnum = rs.getString(1);        // Retrieve the first column value
 phonenum = rs.getString(2);      // Retrieve the first column value
 System.out.println("Employee number = " + empnum +
   "Phone number = " + phonenum);
                                  // Print the column values
}
rs.close();                       // Close the ResultSet                 5
pstmt.close();                    // Close the PreparedStatement         6
```

Figure 13. Using PreparedStatement.executeQuery

You can also use the `PreparedStatement.executeQuery` method for statements that have no parameter markers. When you execute a query many times, you can get better performance by creating the SQL statement as a `PreparedStatement`.

Related tasks:

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 297

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Using CallableStatement methods to call stored procedures

To call stored procedures, you invoke methods in the `CallableStatement` class. The basic steps are:

1. Invoke the `Connection.prepareCall` method to create a `CallableStatement` object.
2. Invoke the `CallableStatement.setXXX` methods to pass values to the input (IN) parameters.
3. Invoke the `CallableStatement.registerOutParameter` method to indicate which parameters are output-only (OUT) parameters, or input and output (INOUT) parameters.
4. Invoke one of the following methods to call the stored procedure:
   `CallableStatement.executeUpdate`
   Invoke this method if the stored procedure does not return result sets.
   `CallableStatement.executeQuery`
   Invoke this method if the stored procedure returns one result set.
   `CallableStatement.execute`
   Invoke this method if the stored procedure returns multiple result sets.
5. If the stored procedure returns result sets, retrieve the result sets. See Retrieve multiple result sets from a stored procedure in a JDBC application.

6. Invoke the `CallableStatement.getXXX` methods to retrieve values from the OUT parameters or INOUT parameters.
7. Invoke the `CallableStatement.close` method to close the `CallableStatement` object when you have finished using that object.

The following code illustrates calling a stored procedure that has one input parameter, four output parameters, and no returned `ResultSets`. The numbers to the right of selected statements correspond to the previously-described steps.

```
int ifcaret;
int ifcareas;
int xsbytes;
String errbuff;
Connection con;
CallableStatement cstmt;
ResultSet rs;
...
cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?,?)");                1
                                  // Create a CallableStatement object
cstmt.setString (1, "DISPLAY THREAD(*)");                               2
                                  // Set input parameter (DB2 command)
cstmt.registerOutParameter (2, Types.INTEGER);                          3
                                  // Register output parameters
cstmt.registerOutParameter (3, Types.INTEGER);
cstmt.registerOutParameter (4, Types.INTEGER);
cstmt.registerOutParameter (5, Types.VARCHAR);
cstmt.executeUpdate();            // Call the stored procedure          4
ifcaret = cstmt.getInt(2);        // Get the output parameter values    6
ifcareas = cstmt.getInt(3);
xsbytes = cstmt.getInt(4);
errbuff = cstmt.getString(5);
cstmt.close();                                                          7
```

Figure 14. Using CallableStatement methods for a stored procedure call with parameter markers

Related tasks:

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 297

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Comparison of driver support for JDBC APIs" on page 376

Handling an SQLException under the DB2 Universal JDBC Driver

As in all Java™ programs, error handling is done using `try/catch` blocks. Methods throw exceptions when an error occurs, and the code in the `catch` block handles those exceptions.

JDBC provides the `SQLException` class for handling errors. All JDBC methods throw an instance of `SQLException` when an error occurs during their execution. According to the JDBC specification, an `SQLException` object contains the following information:

- A `String` object that contains a description of the error, or null
- A `String` object that contains the SQLSTATE, or null

- An int value that contains an error code
- A pointer to the next SQLException, or null

The DB2 Universal JDBC Driver provides a com.ibm.db2.jcc.DB2Diagnosable interface that extends the SQLException class. The DB2Diagnosable interface gives you more information about errors that occur when DB2® is accessed. If the JDBC driver detects an error, DB2Diagnosable gives you the same information as the standard SQLException class. However, if DB2 detects the error, DB2Diagnosable adds the following methods, which give you additional information about the error:

getSqlca
Returns an DB2Sqlca object with the following information:
- An SQL error code
- The SQLERRMC values
- The SQLERRP value
- The SQLERRD values
- The SQLWARN values
- The SQLSTATE

getThrowable
Returns a java.lang.Throwable object that caused the SQLException, or null, if no such object exists.

printTrace
Prints diagnostic information.

The basic steps for handling an SQLException in a JDBC program that runs under the DB2 Universal JDBC Driver are:

1. Give the program access to the com.ibm.db2.jcc.DB2Diagnosable interface and the com.ibm.db2.jcc.DB2Sqlca class. You can fully qualify all references to them, or you can import them:
   ```
   import com.ibm.db2.jcc.DB2Diagnosable;
   import com.ibm.db2.jcc.DB2Sqlca;
   ```
2. Put code that can generate an SQLException in a try block.
3. In the catch block, perform the following steps in a loop:
   a. Test whether you have retrieved the last SQLException. If not, continue to the next step.
   b. Check whether any DB2-only information exists by testing for the existence of a DB2Diagnosable object. If the object exists:
      1) Optional: Invoke the DB2Diagnosable.printTrace method to write all SQLException information to a java.io.PrintWriter object.
      2) Invoke the DB2Diagnosable.getThrowable method to determine whether an underlying java.lang.Throwable caused the SQLException.
      3) Invoke the DB2Diagnosable.getSqlca method to retrieve the DB2Sqlca object.
      4) Invoke the DB2Sqlca.getSqlCode method to retrieve an SQL error code value.
      5) Invoke the DB2Sqlca.getSqlErrmc method to retrieve a string that contains all SQLERRMC values, or invoke the DB2Sqlca.getSqlErrmcTokens method to retrieve the SQLERRMC values in an array.
      6) Invoke the DB2Sqlca.getSqlErrp method to retrieve the SQLERRP value.

7) Invoke the `DB2Sqlca.getSqlErrd` method to retrieve the SQLERRD values in an array.
8) Invoke the `DB2Sqlca.getSqlWarn` method to retrieve the SQLWARN values in an array.
9) Invoke the `DB2Sqlca.getSqlState` method to retrieve the SQLSTATE value.
10) Invoke the `DB2Sqlca.getMessage` method to retrieve error message text from the database server.

c. Invoke the `SQLException.getNextException` method to retrieve the next `SQLException`.

The following code demonstrates how to obtain information from the DB2 version of an `SQLException` that is provided with the DB2 Universal JDBC Driver. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;                      // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable;  // Import packages for DB2       1
import com.ibm.db2.jcc.DB2Sqlca;        // SQLException support
java.io.PrintWriter printWriter;        // For dumping all SQLException
                                        // information
...
try {                                                                   2
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more    3a
                                      // SQLExceptions to process
      //=====> Optional DB2-only error processing
      if (sqle instanceof DB2Diagnosable) {                             3b
                                      // Check if DB2-only information exists
          com.ibm.db2.jcc.DB2Diagnosable diagnosable =
            (com.ibm.db2.jcc.DB2Diagnosable)sqle;
          diagnosable.printTrace (printWriter, "");                     3b1
          java.lang.Throwable throwable =
            diagnosable.getThrowable();                                 3b2
          if (throwable != null) {
            // Extract java.lang.Throwable information
            // such as message or stack trace.
            ...
          }
          DB2Sqlca sqlca = diagnosable.getSqlca();                      3b3
                                        // Get DB2Sqlca object
          if (sqlca != null) {          // Check that DB2Sqlca is not null
            int sqlCode = sqlca.getSqlCode(); // Get the SQL error code 3b4
            String sqlErrmc = sqlca.getSqlErrmc();                      3b5
                                        // Get the entire SQLERRMC
            String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
                                        // You can also retrieve the
                                        // individual SQLERRMC tokens
            String sqlErrp = sqlca.getSqlErrp();                        3b6
                                        // Get the SQLERRP
            int[] sqlErrd = sqlca.getSqlErrd();                         3b7
                                        // Get SQLERRD fields
            char[] sqlWarn = sqlca.getSqlWarn();                        3b8
                                        // Get SQLWARN fields
            String sqlState = sqlca.getSqlState();                      3b9
                                        // Get SQLSTATE
            String errMessage = sqlca.getMessage();                     3b10
                                        // Get error message

            System.err.println ("Server error message: " + errMessage);

            System.err.println ("--------------- SQLCA ---------------");
            System.err.println ("Error code: " + sqlCode);
            System.err.println ("SQLERRMC: " + sqlErrmc);
            for (int i=0; i< sqlErrmcTokens.length; i++) {
              System.err.println ("  token " + i + ": " + sqlErrmcTokens[i]);
            }
```

Figure 15. Processing an SQLException under the DB2 Universal JDBC Driver (Part 1 of 2)

```
            System.err.println ( "SQLERRP: " + sqlErrp );
            System.err.println (
              "SQLERRD(1): " + sqlErrd[0] + "\n" +
              "SQLERRD(2): " + sqlErrd[1] + "\n" +
              "SQLERRD(3): " + sqlErrd[2] + "\n" +
              "SQLERRD(4): " + sqlErrd[3] + "\n" +
              "SQLERRD(5): " + sqlErrd[4] + "\n" +
              "SQLERRD(6): " + sqlErrd[5] );
            System.err.println (
              "SQLWARN1: " + sqlWarn[0] + "\n" +
              "SQLWARN2: " + sqlWarn[1] + "\n" +
              "SQLWARN3: " + sqlWarn[2] + "\n" +
              "SQLWARN4: " + sqlWarn[3] + "\n" +
              "SQLWARN5: " + sqlWarn[4] + "\n" +
              "SQLWARN6: " + sqlWarn[5] + "\n" +
              "SQLWARN7: " + sqlWarn[6] + "\n" +
              "SQLWARN8: " + sqlWarn[7] + "\n" +
              "SQLWARN9: " + sqlWarn[8] + "\n" +
              "SQLWARNA: " + sqlWarn[9] );
            System.err.println ("SQLSTATE: " + sqlState);
                                          // portion of SQLException
          }
        sqle=sqle.getNextException();     // Retrieve next SQLException      3c
      }
}
```

Figure 15. Processing an SQLException under the DB2 Universal JDBC Driver (Part 2 of 2)

Related reference:

- "Error codes issued by the DB2 Universal JDBC Driver" on page 434

Handling an SQLException under the DB2 JDBC Type 2 Driver

As in all Java™ programs, error handling is done using `try/catch` blocks. Methods throw exceptions when an error occurs, and the code in the `catch` block handles those exceptions.

JDBC provides the `SQLException` class for handling errors. All JDBC methods throw an instance of `SQLException` when an error occurs during their execution. According to the JDBC specification, an `SQLException` object contains the following information:

- A `String` object that contains a description of the error, or null
- A `String` object that contains the SQLSTATE, or null
- An `int` value that contains an error code
- A pointer to the next `SQLException`, or null

The basic steps for handling an `SQLException` in a JDBC program that runs under the DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver) are:

1. Put code that can generate an `SQLException` in a `try` block.
2. In the `catch` block, perform the following steps in a loop:
   a. Test whether you have retrieved the last `SQLException`. If not, continue to the next step.
   b. Retrieve error information from the `SQLException`.
   c. Invoke the `SQLException.getNextException` method to retrieve the next `SQLException`.

The following code illustrates a `catch` block that uses the DB2 version of `SQLException` that is provided with the DB2 JDBC Type 2 Driver. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;                      // Import JDBC API package
...
try {
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more    1
      System.out.println("Message:  " + sqle.getMessage());               2
      System.out.println("SQLSTATE: " + sqle.getSQLState());
      System.out.println("SQL error code: " + sqle.getErrorCode());
      sqle=sqle.getNextException();     // Retrieve next SQLException    3
    }
}
```

Figure 16. Processing an SQLException under the DB2 Universal JDBC Driver

Related tasks:

• "Handling an SQLWarning under the DB2 Universal JDBC Driver" on page 287

Handling an SQLWarning under the DB2 Universal JDBC Driver

Unlike SQL errors, SQL warnings do not cause JDBC methods to throw exceptions. Instead, the `Connection`, `Statement`, `PreparedStatement`, `CallableStatement`, and `ResultSet` classes contain `getWarnings` methods, which you need to invoke after you execute SQL statements to determine whether any SQL warnings were generated. Calling `getWarnings` retrieves an `SQLWarning` object. A generic `SQLWarning` object contains the following information:

- A `String` object that contains a description of the warning, or null
- A `String` object that contains the SQLSTATE, or null
- An `int` value that contains an error code
- A pointer to the next `SQLWarning`, or null

Under the DB2 Universal JDBC Driver, like an `SQLException` object, an `SQLWarning` object can also contain DB2®-specific information. The DB2-specific information for an `SQLWarning` object is the same as the DB2-specific information for an `SQLException` object.

The basic steps for retrieving SQL warning information are:

1. Immediately after invoking a method that executes an SQL statement, invoke the `getWarnings` method to retrieve an `SQLWarning` object.
2. Perform the following steps in a loop:
   a. Test whether the `SQLWarning` object is null. If not, continue to the next step.
   b. Invoke the `SQLWarning.getMessage` method to retrieve the warning description.
   c. Invoke the `SQLWarning.getSQLState` method to retrieve the SQLSTATE value.
   d. Invoke the `SQLWarning.getErrorCode` method to retrieve the error code value.
   e. If you want DB2-specific warning information, perform the same steps that you perform to get DB2-specific information for an `SQLException`.

f. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

The following code illustrates how to obtain generic SQLWarning information. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
                                  // Get the result table from the query
sqlwarn = stmt.getWarnings();     // Get any warnings generated          1
while (sqlwarn != null) {         // While there are warnings, get and   2a
                                  // print warning information
  System.out.println ("Warning description: " + sqlwarn.getMessage());   2b
  System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());             2c
  System.out.println ("Error code: " + sqlwarn.getErrorCode());         2d
  sqlwarn=sqlwarn.getNextWarning();     // Get next SQLWarning           2f
}
```

Figure 17. Processing an SQLWarning

Related tasks:

• "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282

Handling an SQLWarning under the DB2 JDBC Type 2 Driver

Unlike SQL errors, SQL warnings do not cause JDBC methods to throw exceptions. Instead, the Connection, Statement, PreparedStatement, CallableStatement, and ResultSet classes contain getWarnings methods, which you need to invoke after you execute SQL statements to determine whether any SQL warnings were generated. Calling getWarnings retrieves an SQLWarning object.

The DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver) generates generic SQLWarning objects. A generic SQLWarning object contains the following information:

- A String object that contains a description of the warning, or null
- A String object that contains the SQLSTATE, or null
- An int value that contains an error code
- A pointer to the next SQLWarning, or null

The basic steps for retrieving SQL warning information are:

1. Immediately after invoking a method that executes an SQL statement, invoke the getWarnings method to retrieve an SQLWarning object.
2. Perform the following steps in a loop:
   a. Test whether the SQLWarning object is null. If not, continue to the next step.
   b. Invoke the SQLWarning.getMessage method to retrieve the warning description.
   c. Invoke the SQLWarning.getSQLState method to retrieve the SQLSTATE value.
   d. Invoke the SQLWarning.getErrorCode method to retrieve the error code value.

e. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

The following code illustrates how to obtain generic SQLWarning information. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
                                  // Get the result table from the query
sqlwarn = stmt.getWarnings();     // Get any warnings generated          1
while (sqlwarn != null) {         // While there are warnings, get and   2a
                                  // print warning information
  System.out.println ("Warning description: " + sqlwarn.getMessage());   2b
  System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());             2c
  System.out.println ("Error code: " + sqlwarn.getErrorCode());         2d
  sqlwarn=sqlwarn.getNextWarning();     // Get next SQLWarning           2f
}
```

Figure 18. Processing an SQLWarning

Related tasks:

- "Handling an SQLException under the DB2 JDBC Type 2 Driver" on page 286

Advanced JDBC application programming concepts

The topics that follow contain more advanced information about writing JDBC applications.

LOBs in JDBC applications with the DB2 Universal JDBC Driver

The DB2 Universal JDBC Driver includes all of the LOB support in the JDBC 2.0 specification. This driver also includes support for LOBs in additional methods and for additional data types.

CLOB data is always sent to the database server as a Unicode stream. The database server converts the data to the target code page.

LOB locator support: The DB2 Universal JDBC Driver can use LOB locators to retrieve data in LOB columns. To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the fullyMaterializeLobData property to false. Properties are discussed in Properties for the DB2® Universal JDBC Driver.

fullyMaterializeLobData has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors. When you fetch data from a DB2 UDB server in the OS/390® or z/OS™ environment using scrollable cursors, JDBC always uses LOB locators to retrieve data from LOB columns.

As in any other language, a LOB locator in a Java application is associated with only one database. You cannot use a single LOB locator to move data between two different databases. To move LOB data between two databases, you need to

materialize the LOB data when you retrieve it from a table in the first database and then insert that data into the table in the second database.

Additional methods supported by the DB2 Universal JDBC Driver: In addition to the methods in the JDBC specification, the DB2 Universal JDBC Driver includes LOB support in the following methods:

- You can specify a BLOB column as an argument of the following `ResultSet` methods to retrieve data from a BLOB column:
  - `getBinaryStream`
  - `getBytes`
- You can specify a CLOB column as an argument of the following `ResultSet` methods to retrieve data from a CLOB column:
  - `getAsciiStream`
  - `getCharacterStream`
  - `getString`
  - `getUnicodeStream`
- You can use the following `PreparedStatement` methods to set the values for parameters that correspond to BLOB columns:
  - `setBytes`
  - `setBinaryStream`
- You can use the following `PreparedStatement` methods to set the values for parameters that correspond to CLOB columns:
  - `setString`
  - `setAsciiStream`
  - `setUnicodeStream`
  - `setCharacterStream`
- You can retrieve the value of a JDBC `CLOB` parameter using the following `CallableStatement` method:
  - `getString`

Restriction on using LOBs with the DB2 Universal JDBC Driver: If you are using Universal Type 2 Connectivity, you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370
- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Comparison of driver support for JDBC APIs" on page 376

Java data types for retrieving or updating LOB column data in JDBC applications

When the deferPrepares property is set to true, and the DB2 Universal JDBC Driver processes a `PreparedStatement.setXXX` call, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

When the JDBC driver cannot immediately determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

Input parameters for BLOB columns:

For input parameters for BLOB columns, or input/output parameters that are used for input to BLOB columns, you can use one of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:

  ```
  cstmt.setBlob(parmIndex, blobData);
  ```

- Use a `CallableStatement.setObject` call that specifies that the target data type is BLOB:

  ```
  byte[] byteData = {(byte)0x1a, (byte)0x2b, (byte)0x3c};
  cstmt.setObject(parmInd, byteData, java.sql.Types.BLOB);
  ```

- Use an input parameter of type of `java.io.ByteArrayInputStream` with a `CallableStatement.setBinaryStream` call. A `java.io.ByteArrayInputStream` object is compatible with a BLOB data type. For this call, you need to specify the exact length of the input data:

  ```
  java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(byteData);
  int numBytes = byteData.length;
  cstmt.setBinaryStream(parmIndex, byteStream, numBytes);
  ```

Output parameters for BLOB columns:

For output parameters for BLOB columns, or input/output parameters that are used for output from BLOB columns, you can use the following technique:

- Use the `CallableStatement.registerOutParameter` call to specify that an output parameter is of type BLOB. Then you can retrieve the parameter value into any variable that has a data type that is compatible with a BLOB data type. For example, the following code lets you retrieve a BLOB value into a byte[] variable:

  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.BLOB);
  cstmt.execute();
  byte[] byteData = cstmt.getBytes(parmIndex);
  ```

Input parameters for CLOB columns:

For input parameters for CLOB columns, or input/output parameters that are used for input to CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:

  ```
  cstmt.setClob(parmIndex, clobData);
  ```

- Use a `CallableStatement.setObject` call that specifies that the target data type is CLOB:

  ```
  String charData = "CharacterString";
  cstmt.setObject(parmInd, charData, java.sql.Types.CLOB);
  ```

- Use one of the following types of stream input parameters:
  - A `java.io.StringReader` input parameter with a `cstmt.setCharacterStream` call:

    ```
    java.io.StringReader reader = new java.io.StringReader(charData);
    cstmt.setCharacterStream(parmIndex, reader, charData.length);
    ```

  - A `java.io.ByteArrayInputStream` parameter with a `cstmt.setAsciiStream` call, for ASCII data:

    ```
    byte[] charDataBytes = charData.getBytes("US-ASCII");
    java.io.ByteArrayInputStream byteStream =
      new java.io.ByteArrayInputStream (charDataBytes);
    cstmt.setAsciiStream(parmIndex, byteStream, charDataBytes.length);
    ```

    For these calls, you need to specify the exact length of the input data.

- Use a String input parameter with a `cstmt.setString` call:

  ```
  cstmt.setString(charData);
  ```

If the length of the data is greater than 32KB, the JDBC driver assigns the CLOB data type to the input data.

- Use a String input parameter with a `cstmt.setObject` call, and specify the target data type as VARCHAR or LONGVARCHAR:

```
cstmt.setObject(parmIndex, charData, java.sql.Types.VARCHAR);
```

If the length of the data is greater than 32KB, the JDBC driver assigns the CLOB data type to the input data.

Output parameters for CLOB columns:

For output parameters for CLOB columns, or input/output parameters that are used for output from CLOB columns, you can use one of the following techniques:

- Use the `CallableStatement.registerOutParameter` call to specify that an output parameter is of type CLOB. Then you can retrieve the parameter value into any variable that has a data type that is compatible with a CLOB data type. For example, the following code lets you retrieve a CLOB value into a String variable:

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.CLOB);
cstmt.execute();
String charData = cstmt.getString(parmIndex);
```

- Use the `CallableStatement.registerOutParameter` call to specify that an output parameter is of type VARCHAR or LONGVARCHAR:

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.VARCHAR);
cstmt.execute();
String charData = cstmt.getString(parmIndex);
```

This technique should be used only if you know that the length of the retrieved data is less than or equal to 32KB. Otherwise, the data is truncated.

Related concepts:

- "LOBs in JDBC applications with the DB2 Universal JDBC Driver" on page 289

Related reference:

- "Java, JDBC, and SQL data types" on page 365

ROWIDs in JDBC with the DB2 Universal JDBC Driver

DB2® UDB for z/OS® and DB2 UDB for iSeries™ support the ROWID data type for a column in a DB2 table. A ROWID is a value that uniquely identifies a row in a table.

You can use the following `ResultSet` methods to retrieve data from a ROWID column:

- `getBytes`
- `getObject`

For `getObject`, the DB2 Universal JDBC Driver returns an instance of the DB2-only class `com.ibm.db2.jcc.DB2RowID`.

You can use the following `PreparedStatement` methods to set a value for a parameter that is associated with a ROWID column:

- `setBytes`
- `setObject`

For setObject, use the DB2-only type com.ibm.db2.jcc.Types.ROWID or an instance of the com.ibm.db2.jcc.DB2RowID class as the target type for the parameter.

Example: Using PreparedStatement.setObject with a com.ibm.db2.jcc.DB2Types.ROWID target type: To set parameter 1, use this form of the SetObject method:

```
ps.setObject(1, bytes[], com.ibm.db2.jcc.DB2Types.ROWID);
```

Example: Using PreparedStatement.setObject with a com.ibm.db2.jcc.DB2RowID target type: Suppose that rwid is an instance of com.ibm.db2.jcc.DB2RowID. To set parameter 1, use this form of the SetObject method:

```
ps.setObject (1, rwid);
```

To call a stored procedure that is defined with a ROWID output parameter, register that parameter to be of the com.ibm.db2.jcc.DB2Types.ROWID type.

Example: Using CallableStatement.registerOutParameter with a com.ibm.db2.jcc.DB2Types.ROWID parameter type: To register parameter 1 of a CALL statement as a com.ibm.db2.jcc.DB2Types.ROWID data type, use this form of the registerOutParameter method:

```
cs.registerOutParameter(1, com.ibm.db2.jcc.DB2Types.ROWID)
```

Related reference:

- "Java, JDBC, and SQL data types" on page 365

Distinct types in JDBC applications

A distinct type is a user-defined data type that is internally represented as a built-in SQL data type. You create a distinct type by executing the SQL statement CREATE DISTINCT TYPE.

In a JDBC program, you can create a distinct type using the executeUpdate method to execute the CREATE DISTINCT TYPE statement. You can also use executeUpdate to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java™ identifiers with data types that correspond to the built-in types on which the distinct types are based.

The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
stmt = con.createStatement();               // Create a Statement object
stmt.executeUpdate(
  "CREATE DISTINCT TYPE SHOESIZE AS INTEGER");
                                            // Create distinct type
stmt.executeUpdate(
  "CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)");
                                            // Create table with distinct type
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000010', 6)");                  // Insert a row
rs=stmt.executeQuery("SELECT EMPNO, EMP_SHOE_SIZE FROM EMP_SHOE);
                                            // Create ResultSet for query
while (rs.next()) {
  empNumVar = rs.getString(1);              // Get employee number
  shoeSizeVar = rs.getInt(2);               // Get shoe size (use int
                                            // because underlying type
                                            // of SHOESIZE is INTEGER)
  System.out.println("Employee number = " + empNumVar +
    " Shoe size = " + shoeSizeVar);
}
rs.close();                                 // Close ResultSet
stmt.close();                               // Close Statement
```

Figure 19. Creating and using a distinct type

Related reference:

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

Savepoints in JDBC applications

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The DB2 Universal JDBC Driver supports the following methods for using savepoints:

`Connection.setSavepoint()` or `Connection.setSavepoint(String` *`name`*`)`
Sets a savepoint. These methods return a `Savepoint` object that is used in later `releaseSavepoint` or `rollback` operations.

When you execute either of these methods, DB2® executes the form of the SAVEPOINT statement that includes ON ROLLBACK RETAIN CURSORS.

`Connection.releaseSavepoint(Savepoint` *`savepoint`*`)`
Releases the specified savepoint, and all subsequently established savepoints.

`Connection.rollback(Savepoint` *`savepoint`*`)`
Rolls back work to the specified savepoint.

`DatabaseMetaData.supportsSavepoints()`
Indicates whether a data source supports savepoints.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
con.setAutoCommit(false);                    // set autocommit OFF
stmt = con.createStatement();                // Create a Statement object
stmt.executeUpdate(
  "CREATE DISTINCT TYPE SHOESIZE AS INTEGER");
                                             // Create distinct type
con.commit();                                // Commit the create
stmt.executeUpdate(
  "CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)");
                                             // Create table with distinct type
con.commit();                                // Commit the create
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000010', 6)");                   // Insert a row
Savepoint savept = con.setSavepoint();       // Create a savepoint
...
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000020', 10)");                  // Insert another row
conn.rollback(savept);                       // Roll back work to the point
                                             //  after the first insert
...
con.releaseSavepoint(savept);                // Release the savepoint
stmt.close();                                // Close the Statement
```

Figure 20. Setting, rolling back to, and releasing a savepoint in a JDBC application

Related tasks:

- "Committing or rolling back JDBC transactions" on page 275

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Retrieving identity column values in JDBC applications

An identity column is a DB2® table column that provides a way for DB2 to automatically generate a numeric value for each row. You define an identity column in a CREATE TABLE or ALTER TABLE statement by specifying the AS IDENTITY clause when you define a column that has an exact numeric type with a scale of 0 (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types).

If you are using the DB2 Universal JDBC Driver, you can retrieve identity columns from a DB2 table using JDBC 3.0 methods. In a JDBC program, identity columns are known as automatically generated keys. To enable retrieval of automatically generated keys from a table, you need to indicate when you insert rows that you will want to retrieve automatically generated key values. You do that by setting a flag in a `Connection.prepareStatement`, `Statement.executeUpdate`, or `Statement.execute` method call. The statement that is executed must be an INSERT statement or an INSERT within SELECT statement. Otherwise, the JDBC driver ignores the parameter that sets the flag.

To retrieve automatically generated keys from a DB2 table, you need to perform these steps:

1. Use one of the following methods to indicate that you want to return automatically generated keys:

- If you plan to use the PreparedStatement.executeUpdate method to insert rows, invoke one of these forms of the Connection.prepareStatement method to create a PreparedStatement object:

  Use this form for a table on any database server that supports identity columns:

  ```
  Connection.prepareStatement(sql-statement,
    Statement.RETURN_GENERATED_KEYS);
  ```

  Use this form only for a table on any database server that supports identity columns and INSERT within SELECT:

  ```
  Connection.prepareStatement(sql-statement, String [] columnNames);
  ```
- If you use the Statement.executeUpdate method to insert rows, invoke one of these form of the Statement.executeUpdate method:

  Use this form for a table on any database server that supports identity columns:

  ```
  Statement.executeUpdate(sql-statement, Statement.RETURN_GENERATED_KEYS);
  ```

  Use this form only for a table on any database server that supports identity columns and INSERT within SELECT:

  ```
  Statement.executeUpdate(sql-statement, String [] columnNames);
  ```
- If you use the Statement.execute method to insert rows, invoke one of these forms of the Statement.execute method:

  Use this form for a table on any database server that supports identity columns:

  ```
  Statement.execute(sql-statement, Statement.RETURN_GENERATED_KEYS);
  ```

  Use this form only for a table on any database server that supports identity columns and INSERT within SELECT:

  ```
  Statement.execute(sql-statement, String [] columnNames);
  ```

2. Invoke the PreparedStatement.getGeneratedKeys method or the Statement.getGeneratedKeys method to retrieve a ResultSet object that contains the automatically generated key values.

   The data type of the automatically generated keys in the ResultSet is DECIMAL, regardless of the data type of the corresponding column.

The following code creates a table with an identity column, inserts rows into the table, and retrieves automatically generated key values for the identity column. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal iDColVar;
...
stmt = con.createStatement();         // Create a Statement object

stmt.executeUpdate(
  "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)");
                                      // Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE " +                          1
  "VALUES ('000010', "5555")",        // Insert a row
  Statement.RETURN_GENERATED_KEYS);   // Indicate you want automatically
                                      // generated keys
rs = stmt.getGeneratedKeys();         // Retrieve the automatically       2
                                      // generated key value in a ResultSet.
                                      // Only one row is returned.
                                      // Create ResultSet for query
while (rs.next()) {
  idColVar = rs.getBigDecimal(1);     // Get automatically generated key
                                      // value
  System.out.println("automatically generated key value = " + idColVar);
}
rs.close();                           // Close ResultSet
stmt.close();                         // Close Statement
```

Figure 21. Retrieving automatically generated keys

Related concepts:

- "Identity Columns" on page 670

Related tasks:

- "Using the PreparedStatement.executeUpdate method to update data in DB2 tables" on page 279
- "Using the Statement.executeUpdate method to create and modify DB2 objects" on page 277

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Retrieving multiple result sets from a stored procedure in a JDBC application

If you call a stored procedure that returns result sets, you need to include code to retrieve the result sets. The steps that you take depend on whether you know how many result sets are returned, and whether you know the contents of those result sets.

Retrieving a known number of result sets:

To retrieve result sets when you know the number of result sets and their contents, follow these steps:

1. Invoke the `Statement.execute` method or `PreparedStatement.execute` method to call the stored procedure. Use `PreparedStatement.execute` if the stored procedure has input parameters.
2. Invoke the `getResultSet` method to obtain the first result set, which is in a `ResultSet` object.

3. In a loop, position the cursor using the `next` method, and retrieve data from each column of the current row of the `ResultSet` object using `getXXX` methods.
4. If there are *n* result sets, repeat the following steps *n*-1 times:
   a. Invoke the `getMoreResults` method to close the current result set and point to the next result set.
   b. Invoke the `getResultSet` method to obtain the next result set, which is in a `ResultSet` object.
   c. In a loop, position the cursor using the `next` method, and retrieve data from each column of the current row of the `ResultSet` object using `getXXX` methods.

The following code illustrates retrieving two result sets. The first result set contains an INTEGER column, and the second result set contains a CHAR column. The numbers to the right of selected statements correspond to the previously-described steps.

```
CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...
cstmt.execute();                          // Call the stored procedure       1
rs = cstmt.getResultSet();                // Get the first result set        2
while (rs.next()) {                       // Position the cursor             3
 i = rs.getInt(1);                        // Retrieve current result set value
 System.out.println("Value from first result set = " + i);
                                          // Print the value
}
cstmt.getMoreResults();                   // Point to the second result set  4a
                                          // and close the first result set
rs = cstmt.getResultSet();                // Get the second result set       4b
while (rs.next()) {                       // Position the cursor             4c
 s = rs.getString(1);                     // Retrieve current result set value
 System.out.println("Value from second result set = " + s);
                                          // Print the value
}
rs.close();                               // Close the result set
cstmt.close();                            // Close the statement
```

Figure 22. Retrieving known result sets from a stored procedure

Retrieving an unknown number of result sets:

To retrieve result sets when you do not know the number of result sets or their contents, you need to retrieve `ResultSets`, until no more `ResultSets` are returned. For each `ResultSet`, use `ResultSetMetaData` methods to determine its contents. See Use ResultSetMetaData to learn about a ResultSet for more information on determining the contents of a `ResultSet`.

After you call a stored procedure, follow these basic steps to retrieve the contents of an unknown number of result sets.

1. Check the value that was returned from the `execute` statement that called the stored procedure. If the returned value is `true`, there is at least one result set, so you need to go to the next step.
2. Repeat the following steps in a loop:
   a. Invoke the `getResultSet` method to obtain a result set, which is in a `ResultSet` object. Invoking this method closes the previous result set.

b. Process the ResultSet, as shown in Use ResultSetMetaData to learn about a ResultSet.
c. Invoke the getMoreResults method to determine whether there is another result set. If getMoreResults returns true, go to step 2a on page 298 to get the next result set.

The following code illustrates retrieving result sets when you do not know the number of result sets or their contents. The numbers to the right of selected statements correspond to the previously-described steps.

```
CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {                  // Test for result sets       1
 ResultSet rs = cstmt.getResultSet();       // Get a result set           2a
 ...                                     // process ResultSet
 resultsAvailable = cstmt.getMoreResults(); // Check for next result set  2c
                                            // (Also closes the
                                            // previous result set)
}
```

Figure 23. Retrieving unknown result sets from a stored procedure

Keeping result sets open:

In Figure 23, invocation of getMoreResults() closes the ResultSet object that is returned by the previous invocation of getResultSet. However, if you are using the DB2 Universal JDBC Driver, you can invoke the JDBC 3 form of getMoreResults, which has a parameter that determines whether the current ResultSet or previously-opened ResultSets are closed. This form of getMoreResults requires JDK 1.4 or later.

You can specify one of these constants:

Statement.KEEP_CURRENT_RESULT
Checks for the next ResultSet, but does not close the current ResultSet.

Statement.CLOSE_CURRENT_RESULT
Checks for the next ResultSet, and closes the current ResultSet.

Statement.CLOSE_ALL_RESULTS
Closes all ResultSets that were previously kept open.

For example, the code in Figure 24 on page 300 keeps all ResultSets open until the final ResultSet has been retrieved, and then closes all ResultSets.

```
CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {                  // Test for result sets
 ResultSet rs = cstmt.getResultSet();       // Get a result set
 ...                                       // process ResultSet
 resultsAvailable = cstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
                                            // Check for next result set
                                            // but do not close
                                            // previous result set
}
resultsAvailable = cstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
                                            // Close the result sets
```

Figure 24. Keeping retrieved stored procedure result sets open

Related tasks:

- "Using ResultSetMetaData to learn about a ResultSet" on page 300

Using ResultSetMetaData to learn about a ResultSet

Previous discussions of retrieving data from a table or stored procedure result set assumed that you know the number of columns and data types of the columns in the table or result set. This is not always the case, especially when you are retrieving data from a remote data source. When you write programs that retrieve unknown ResultSets, you need to use ResultSetMetaData methods to determine the characteristics of the ResultSets before you can retrieve data from them.

ResultSetMetaData methods provide the following types of information:

- The number of columns in a ResultSet
- The qualifier for the underlying table of the ResultSet
- Information about a column, such as the data type, length, precision, scale, and nullability
- Whether a column is read-only

After you invoke the executeQuery method to generate a ResultSet for a query on a table, follow these basic steps to determine the contents of the ResultSet:

1. Invoke the getMetaData method on the ResultSet object to create a ResultSetMetaData object.
2. Invoke the getColumnCount method to determine how many columns are in the ResultSet.
3. For each column in the ResultSet, execute ResultSetMetaData methods to determine column characteristics.

   The results of ResultSetMetaData.getColumnName for the same table definition might differ, depending on the data source. However, the returned information correctly reflects the column name information that is stored in the DB2® catalog for that data source.

For example, the following code demonstrates how to determine the data types of all the columns in the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmtadta;
int colCount
int mtadtaint;
int i;
String colName;
String colType;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
                                  // Get the ResultSet from the query
rsmtadta = rs.getMetaData();      // Create a ResultSetMetaData object  1
colCount = rsmtadta.getColumnCount();                                  2
                                  // Find number of columns in EMP
for (i=1; i<= colCount; i++) {                                         3
 colName = rsmtadta.getColumnName();    // Get column name
 colType = rsmtadta.getColumnTypeName();
                                        // Get column data type
 System.out.println("Column = " + colName +
  " is data type " + colType);
                                        // Print the column value
}
```

Figure 25. Using ResultSetMetaData methods to get information about a ResultSet

Related tasks:

- "Using CallableStatement methods to call stored procedures" on page 281
- "Using the Statement.executeQuery method to retrieve data from DB2 tables" on page 277

Using DatabaseMetaData to learn about a data source

The `DatabaseMetaData` interface contains methods that retrieve information about a data source. These methods are useful when you write generic applications that can access various data sources. In these types of applications, you need to test whether a data source can handle various database operations before you execute them. For example, you need to determine whether the driver at a data source is at the JDBC 2.0 level before you invoke JDBC 2.0 methods against that driver.

`DatabaseMetaData` methods provide the following types of information:

- Features that the data source supports, such as the ANSI SQL level
- Specific information about the data source, such as the driver level
- Limits, such as the maximum number of columns that an index can have
- Whether the data source supports data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE)
- Lists of objects at the data source, such as tables, indexes, or procedures
- Whether the data source supports various JDBC 2.0 functions, such as batch updates or scrollable `ResultSets`

If your application connects to a DB2® UDB for z/OS™ or OS/390® server, a number of stored procedures need to be installed on that server before you can invoke some `DatabaseMetaData` methods that require DB2 catalog information. The stored procedures are:

- SQLCOLPRIVILEGES
- SQLCOLUMNS
- SQLFOREIGNKEYS

- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS

For DB2 UDB for OS/390 and z/OS, Version 7 or DB2 UDB for OS/390, Version 6, the stored procedures are shipped in PTFs. The PTFs are orderable through normal service channels using the following PTF numbers:

Table 33. PTFs for DB2 Universal Database for z/OS and OS/390

DB2 Universal Database for z/OS and OS/390 Version	PTF number
Version 6	UQ72081 and UQ72082
Version 7	UQ72083

Ask your DB2 UDB for z/OS system administrator whether these stored procedures are installed.

To invoke `DatabaseMetaData` methods, you need to perform these basic steps:

1. Create a `DatabaseMetaData` object by invoking the `getMetaData` method on the connection.
2. Invoke `DatabaseMetaData` methods to get information about the data source.
3. If the method returns a `ResultSet`:
   a. In a loop, position the cursor using the `next` method, and retrieve data from each column of the current row of the `ResultSet` object using `getXXX` methods.
   b. Invoke the `close` method to close the `ResultSet` object.

For example, the following code demonstrates how to use `DatabaseMetaData` methods to determine the driver version and get a list of the stored procedures that are available at the data source. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
DatabaseMetaData dbmtadta;
ResultSet rs;
int mtadtaint;
String procSchema;
String procName;
...
dbmtadta = con.getMetaData();     // Create the DatabaseMetaData object  1
mtadtaint = dmtadta.getDriverVersion();                                  2
                                  // Check the driver version
System.out.println("Driver version: " + mtadtaint);
rs = dbmtadta.getProcedures(null, null, "%");
                                  // Get information for all procedures
while (rs.next()) {               // Position the cursor                 3a
 procSchema = rs.getString("PROCEDURE_SCHEM");
                                  // Get procedure schema
 procName = rs.getString("PROCEDURE_NAME");
                                  // Get procedure name
 System.out.println(procSchema + "." + procName);
                                  // Print the qualified procedure name
}
rs.close();                       // Close the ResultSet                 3b
```

Figure 26. Using DatabaseMetaData methods to get information about a data source

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Comparison of driver support for JDBC APIs" on page 376

Using ParameterMetaData to learn about parameters in a PreparedStatement

The DB2 Universal JDBC Driver includes support for the `ParameterMetaData` interface. The `ParameterMetaData` interface contains methods that retrieve information about the parameter markers in a `PreparedStatement` object.

`ParameterMetaData` methods provide the following types of information:

- The data types of parameters, including the precision and scale of decimal parameters.
- The parameters' database-specific type names. For parameters that correspond to table columns that are defined with distinct types, these names are the distinct type names.
- Whether parameters are nullable.
- Whether parameters are input or output parameters.
- Whether the values of a numeric parameter can be signed.
- The fully-qualified Java™ class name that `PreparedStatement.setObject` uses when it sets a parameter value.

To invoke `ParameterMetaData` methods, you need to perform these basic steps:

1. Invoke the `Connection.prepareStatement` method to create a `PreparedStatement` object.
2. Invoke the `PreparedStatement.getParameterMetaData` method to retrieve a `ParameterMetaData` object.
3. Invoke `ParameterMetaData.getParameterCount` to determine the number of parameters in the `PreparedStatement`.
4. Invoke `ParameterMetaData` methods on individual parameters.

For example, the following code demonstrates how to use ParameterMetaData methods to determine the number and data types of parameters in an SQL UPDATE statement. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
int sqlType;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                  // Create a PreparedStatement object  1
pmtadta = pstmt.getParameterMetaData();                                  2
                                  // Create a ParameterMetaData object
mtadtacnt = pmtadta.getParameterCount();                                 3
                                  // Determine the number of parameters
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
  sqlType = pmtadta.getParameterType(i);                                 4
                                  // Get SQL type for each parameter
 System.out.println("SQL type of parameter " + i " is " + sqlType);
}
...
pstmt.close();                      // Close the PreparedStatement
```

Figure 27. Using ParameterMetaData methods to get information about a PreparedStatement

Related reference:

- "Comparison of driver support for JDBC APIs" on page 376

Making batch updates in JDBC applications

The JDBC drivers that support JDBC 2.0 and above support batch updates. With batch updates, instead of updating rows of a DB2® table one at a time, you can direct JDBC to execute a group of updates at the same time. Statements that can be included in the same batch of updates are known as *batchable* statements.

If a statement has input parameters or host expressions, you can include that statement only in a batch that has other instances of the same statement. This type of batch is known as a *homogeneous batch*. If a statement has no input parameters, you can include that statement in a batch only if the other statements in the batch have no input parameters or host expressions. This type of batch is known as a *heterogeneous batch*. Two statements that can be included in the same batch are known as *batch compatible*.

Use the following Statement methods for creating, executing, and removing a batch of SQL updates:

- addBatch
- executeBatch
- clearBatch

Use the following PreparedStatement and CallableStatement method for creating a batch of parameters so that a single statement can be executed multiple times in a batch, with a different set of parameters for each execution.

- addBatch

To make batch updates using several statements with no input parameters, follow these basic steps:

1. Disable `AutoCommit` for the `Connection` object.
2. Invoke the `createStatement` method to create a `Statement` object.
3. For each SQL statement that you want to execute in the batch, invoke the `addBatch` method.
4. Invoke the `executeBatch` method to execute the batch of statements.
5. Check for errors. If no errors occurred:
   a. Get the number of rows that were affect by each SQL statement from the array that the `executeBatch` invocation returns. This number does not include rows that were affected by triggers or by referential integrity enforcement.
   b. Invoke the `commit` method to commit the changes.

To make batch updates using a single statement with several sets of input parameters, follow these basic steps:

1. Disable `AutoCommit` for the `Connection` object.
2. Invoke the `prepareStatement` method to create a `PreparedStatement` object for the SQL statement with input parameters.
3. For each set of input parameter values:
   a. Execute `setXXX` methods to assign values to the input parameters.
   b. Invoke the `addBatch` method to add the set of input parameters to the batch.
4. Invoke the `executeBatch` method to execute the statements with all sets of parameters.
5. Check for errors. If no errors occurred:
   a. Get the number of rows that were updated by each execution of the SQL statement from the array that the `executeBatch` invocation returns.
   b. Invoke the `commit` method to commit the changes.

Example of a batch update: In the following code fragment, two sets of parameters are batched. An UPDATE statement that takes two input parameters is then executed twice, once with each set of parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```
try {
...
  connection con.setAutoCommit(false);                    1
  PreparedStatement prepStmt = con.prepareStatement(
    "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");            2
  prepStmt.setString(1,mgrnum1);                          3a
  prepStmt.setString(2,deptnum1);
  prepStmt.addBatch();                                    3b

  prepStmt.setString(1,mgrnum2);
  prepStmt.setString(2,deptnum2);
  prepStmt.addBatch();
  int [] numUpdates=prepStmt.executeBatch();              4
  for (int i=0; i < numUpdates.length; i++) {             5a
    if (numUpdates[i] == -2)
      System.out.println("Execution " + i +
        ": unknown number of rows updated");
    else
      System.out.println("Execution " + i +
        "successful: " numUpdates[i] + " rows updated");
  }
  con.commit();                                           5b
} catch(BatchUpdateException b) {
  // process BatchUpdateException
}
```

Figure 28. Performing a batch update

Related tasks:

- "Committing or rolling back JDBC transactions" on page 275

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426

Retrieving information from a BatchUpdateException

When an error occurs during execution of a statement in a batch, processing continues. However, `executeBatch` throws a `BatchUpdateException`. A `BatchUpdateException` object contains the following items:

- A `String` object that contains a description of the error, or `null`.
- A `String` object that contains the SQLSTATE for the failing SQL statement, or `null`
- An integer value that contains the error code, or zero
- An integer array of update counts for SQL statements in the batch, or `null`
- A pointer to an `SQLException` object, or `null`

One `BatchUpdateException` is thrown for the entire batch. At least one `SQLException` object is chained to the `BatchUpdateException` object. The `SQLException` objects are chained in the same order as the corresponding statements were added to the batch. To help you match `SQLException` objects to statements in the batch, the error description field for each `SQLException` object begins with this string:

`Error for batch element #n:`

n is the number of the statement in the batch.

To retrieve information from the `BatchUpdateException`, follow these steps:

1. Use the BatchUpdateException.getUpdateCounts method to determine the number of rows that each SQL statement updated. getUpdateCounts returns -2 if the number of updated rows cannot be determined, or -3 if an error occurred during an update.
2. Use SQLException methods getMessage, getSQLState, and getErrorCode to retrieve the description of the error, the SQLSTATE, and the error code for the first error.
3. Use the BatchUpdateException.getNextException method to get a chained SQLException.
4. In a loop, execute the getMessage, getSQLState, getErrorCode, and getNextException method calls to obtain information about an SQLException and get the next SQLException.

Example of obtaining information from a BatchUpdateException: The following code fragment demonstrates how to obtain the fields of a BatchUpdateException and the chained SQLException objects. The numbers to the right of selected statements correspond to the previously-described steps.

```
try {
  // Batch updates
} catch(BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();             1
    for (int i = 0; i < updateCounts.length; i++) {
      System.err.println("  Statement " + i + ":" + updateCounts[i]);
    }
    System.err.println(" Message: " + buex.getMessage());     2
    System.err.println(" SQLSTATE: " + buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();                3
    while (ex != null) {                                      4
      System.err.println("SQL exception:");
      System.err.println(" Message: " + ex.getMessage());
      System.err.println(" SQLSTATE: " + ex.getSQLState());
      System.err.println(" Error code: " + ex.getErrorCode());
      ex = ex.getNextException();
    }
}
```

Figure 29. Retrieving a BatchUpdateException fields

To obtain information about warnings, use the Statement.getWarnings method on the object on which you ran the executeBatch method. You can then retrieve an error description, SQLSTATE, and error code for each SQLWarning object.

Restrictions on executing statements in a batch:

- If you try to execute a SELECT statement in a batch, a BatchUpdateException is thrown.
- A CallableStatement object that you execute in a batch can contain output parameters. However, you cannot retrieve the values of the output parameters. If you try to do so, a BatchUpdateException is thrown.
- You cannot retrieve ResultSet objects from a CallableStatement object that you execute in a batch. A BatchUpdateException is not thrown, but the getResultSet method invocation returns a null value.

Related tasks:

- "Making batch updates in JDBC applications" on page 304

Characteristics of a JDBC ResultSet under the DB2 Universal JDBC Driver

In addition to moving forward, one row at a time, through a `ResultSet`, you might want to do the following things:

- Move backward or go directly to a specific row
- Update or delete rows of a `ResultSet`
- Leave the `ResultSet` open after a COMMIT

The following terms describe characteristics of a `ResultSet`:

scrollability
Whether the cursor can move forward, backward, or to a specific row.

updatability
Whether the cursor can be used to update or delete rows. This characteristic does not apply to a `ResultSet` that is returned from a stored procedure, because a stored procedure `ResultSet` cannot be updated.

holdability
Whether the cursor stays open after a COMMIT.

A scrollable `ResultSet` in JDBC is equivalent to the result table of a DB2® cursor that is declared as SCROLL. A scrollable cursor can be *insensitive* or *sensitive*. Insensitive means that changes to the underlying table after the cursor is opened are not visible to the cursor. Insensitive cursors are read-only. Sensitive means the following things:

- Changes that the cursor makes to the underlying table are always visible to the cursor.
- Changes that are made by other means to the underlying table *can* be visible to the cursor. In DB2, if the rows are fetched with FETCH INSENSITIVE, changes that are made by other means are not visible to the cursor. If the rows are fetched with FETCH SENSITIVE, changes that are made by other means are visible to the cursor. In JDBC, calling the `refreshRow` method before calling `getXXX` methods has the same effect as FETCH SENSITIVE.

A JDBC `ResultSet` can also be *static* or *dynamic*, if the database server supports both attributes. You determine whether scrollable cursors in a program are static or dynamic by setting the cursorSensitivity property. See Properties for the DB2 Universal JDBC Driver for more information about the cursorSensitivity property.

If a JDBC `ResultSet` is static, the size of the result table and the order of the rows in the result table do not change after the cursor is opened. This means that you cannot insert into a result table, and if you delete a row of a result table, a delete hole occurs.You can test whether the current row is a delete hole by using the `rowDeleted` method. See Comparison of driver support for JDBC APIs for a complete list of the methods that are supported for `ResultSets`.

Related tasks:

- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 309

Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications

To specify scrollability, updatability, and holdability for a `ResultSet`, you need to follow these steps:

1. If the SELECT statement that defines the `ResultSet` has no input parameters, invoke the `createStatement` method to create a `Statement` object. Otherwise, invoke the `prepareStatement` method to create a `PreparedStatement` object.

   You need to specify forms of the `createStatement` or `prepareStatement` methods that include the *resultSetType*, *resultSetConcurrency*, or *resultSetHoldability* parameters.

   The form of the `createStatement` method that supports scrollability and updatability is:

   ```
   createStatement(int resultSetType, int resultSetConcurrency);
   ```

   The form of the `createStatement` method that supports scrollability, updatability, and holdability is:

   ```
   createStatement(int resultSetType, int resultSetConcurrency,
    int resultSetHoldability);
   ```

   The form of the `prepareStatement` method that supports scrollability and updatability is:

   ```
   prepareStatement(String sql, int resultSetType,
    int resultSetConcurrency);
   ```

   The form of the `prepareStatement` method that supports scrollability, updatability, and holdability is:

   ```
   prepareStatement(String sql, int resultSetType,
    int resultSetConcurrency, int resultSetHoldability);
   ```

   See Table 34 for a list of valid values for *resultSetType* and *resultSetConcurrency*.

Table 34. Valid combinations of resultSetType and resultSetConcurrency for scrollable ResultSets

resultSetType **value**	*resultSetConcurrency* **value**
`TYPE_FORWARD_ONLY`	`CONCUR_READ_ONLY`
`TYPE_FORWARD_ONLY`	`CONCUR_UPDATABLE`
`TYPE_SCROLL_INSENSITIVE`	`CONCUR_READ_ONLY`
`TYPE_SCROLL_SENSITIVE`	`CONCUR_READ_ONLY`
`TYPE_SCROLL_SENSITIVE`	`CONCUR_UPDATABLE`

   resultSetHoldability has two possible values: `HOLD_CURSORS_OVER_COMMIT` and `CLOSE_CURSORS_AT_COMMIT`. Either of these values can be specified with any valid combination of *resultSetConcurrency* and *resultSetHoldability*. The value that you set overrides the default holdability for the connection.

   Restriction: If the `ResultSet` is scrollable, and the `ResultSet` is used to select columns from a table on a DB2 UDB for Linux, UNIX, and Windows server, the SELECT statement that defines the `ResultSet` cannot select columns with the following data types:
   - LONG VARCHAR
   - LONG VARGRAPHIC
   - DATALINK
   - BLOB
   - CLOB

- A distinct type that is based on any of the previous data types in this list
- A structured type

2. If the SELECT statement has input parameters, invoke `setXXX` methods to pass values to the input parameters.
3. Invoke the `executeQuery` method to obtain the result table from the SELECT statement in a `ResultSet` object.
4. For each row that you want to access:
   a. Position the cursor using one of the methods list in Table 35.

Table 35. ResultSet methods for positioning a scrollable cursor

Method	Positions the cursor
`first()`	On the first row of the `ResultSet`
`last()`	On the last row of the `ResultSet`
`next()`[1]	On the next row of the `ResultSet`
`previous()`[2]	On the previous row of the `ResultSet`
`absolute(int n)`[3]	If $n>0$, on row n of the `ResultSet`. If $n<0$, and m is the number of rows in the `ResultSet`, on row $m+n+1$ of the `ResultSet`.
`relative(int n)`[4,5]	If $n>0$, on the row that is n rows after the current row. If $n<0$, on the row that is n rows before the current row. If $n=0$, on the current row.
`afterLast()`	After the last row in the `ResultSet`
`beforeFirst()`	Before the first row in the `ResultSet`

Notes:

1. If the cursor is before the first row of the `ResultSet`, this method positions the cursor on the first row.
2. If the cursor is after the last row of the `ResultSet`, this method positions the cursor on the last row.
3. If the absolute value of n is greater than the number of rows in the result set, this method positions the cursor after the last row if n is positive, or before the first row if n is negative.
4. The cursor must be on a valid row of the `ResultSet` before you can use this method. If the cursor is before the first row or after the last throw, the method throws an `SQLException`.
5. Suppose that m is the number of rows in the `ResultSet` and x is the current row number in the `ResultSet`. If $n>0$ and $x+n>m$, the driver positions the cursor after the last row. If $n<0$ and $x+n<1$, the driver positions the cursor before the first row.

   b. If you need to know the current cursor position, use the `getRow`, `isFirst`, `isLast`, `isBeforeFirst`, or `isAfterLast` method to obtain this information.
   c. If you specified a *resultSetType* value of `TYPE_SCROLL_SENSITIVE` in step 1 on page 309, and you need to see the latest values of the current row, invoke the `refreshRow` method.

      Recommendation: Because refreshing the rows of a `ResultSet` can have a detrimental effect on the performance of your applications, you should invoke `refreshRow` *only* when you need to see the latest data.
   d. Perform one or more of the following operations:
      - To retrieve data from each column of the current row of the `ResultSet` object, use `getXXX` methods.
      - To update the current row from the underlying table, use `updateXXX` methods to assign column values to the current row of the `ResultSet`.

Then use `updateRow` to update the corresponding row of the underlying table. If you decide that you do not want to update the underlying table, invoke the `cancelRowUpdates` method instead of the `updateRow` method.

The *resultSetConcurrency* value for the `ResultSet` must be `CONCUR_UPDATABLE` for you to use these methods.

- To delete the current row from the underlying table, use the `deleteRow` method. Invoking `deleteRow` causes the driver to replace the current row of the `ResultSet` with a hole.

  The *resultSetConcurrency* value for the `ResultSet` must be `CONCUR_UPDATABLE` for you to use this method.

5. Invoke the `close` method to close the `ResultSet` object.
6. Invoke the `close` method to close the `Statement` or `PreparedStatement` object.

For example, the following code demonstrates how to retrieve all rows from the employee table in reverse order, and update the phone number for employee number ″000010″. The numbers to the right of selected statements correspond to the previously-described steps.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);                 1
                                        // Create a Statement object
                                        // for a scrollable, updatable
                                        // ResultSet
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO");
                                       // Create the ResultSet          3
rs.afterLast();                        // Position the cursor at the end of
                                       // the ResultSet                 4a
while (rs.previous()) {                // Position the cursor backward
 s = rs.getString("EMPNO");            // Retrieve the employee number  4d
                                       // (column 1 in the result
                                       // table)
 System.out.println("Employee number = " + s);
                                       // Print the column value
 if (s.compareTo("000010") == 0) {     // Look for employee 000010
  updateString("PHONENO","4657");      // Update their phone number
  updateRow();                         // Update the row
 }
}
rs.close();                            // Close the ResultSet           5
stmt.close();                          // Close the Statement           6
```

Figure 30. Using a scrollable cursor

Creating and deploying DataSource objects

JDBC versions starting with version 2.0 provide the `DataSource` interface for connecting to a data source. Using the `DataSource` interface is the preferred way to connect to a data source. Using the `DataSource` interface involves two parts:

- Creating and deploying `DataSource` objects. This is usually done by a system administrator, using a tool such as WebSphere® Application Server.
- Using the `DataSource` objects to create a connection. This is done in the application program.

This topic contains information that you need if you create and deploy the DataSource objects yourself.

The DB2 Universal JDBC Driver provides the following DataSource implementations:

- com.ibm.db2.jcc.DB2SimpleDataSource, which does not support connection pooling. You can use this implementation with Universal Type 2 Connectivity or Universal Type 4 Connectivity.

The DB2® JDBC Type 2 Driver provides the following DataSource implementations:

- COM.ibm.db2.jdbc.DB2DataSource, which has built-in support for connection pooling. With this implementation, connection pooling is handled internally and is transparent to the application.
- COM.ibm.db2.jdbc.DB2XADataSource, which does not have built-in support for distributed transactions and connection pooling. With this implementation, you must manage the distributed transactions and connection pooling yourself, either by writing your own code or by using a tool such as WebSphere Application Server.

When you create and deploy a DataSource object, you need to perform these tasks:

1. Create an instance of the appropriate DataSource implementation.
2. Set the properties of the DataSource object.
3. Register the object with the Java™ Naming and Directory Interface (JNDI) naming service.

The example in Figure 31 shows how to perform these tasks.

```
import java.sql.*;        // JDBC base
import javax.naming.*;    // JNDI Naming Services
import javax.sql.*;       // JDBC 2.0 standard extension APIs
import com.ibm.db2.jcc.*; // DB2 implementation of JDBC 2.0
                          // standard extension APIs

DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource(); 1

db2ds.setDatabaseName("db2loc1");                                      2
db2ds.setDescription("Our Sample Database");
db2ds.setUser("john");
db2ds.setPassword("db2");
:
:
Context ctx=new InitialContext();                                      3
Ctx.bind("jdbc/sampledb",db2ds);                                       4
```

Figure 31. Example of creating and deploying a DataSource object

1. Creates an instance of the DB2SimpleDataSource class.
2. This statement and the next three statements set values for properties of this DB2SimpleDataSource object.
3. Creates a context for use by JNDI.
4. Associates DBSimple2DataSource object db2ds with the logical name jdbc/sampledb. An application that uses this object can refer to it by the name jdbc/sampledb.

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

DB2 Universal JDBC Driver client reroute support

Failover is the ability of a server to take over operations when another server fails. DB2 Universal JDBC Driver client reroute support provides failover support in a DB2® UDB for Linux, UNIX® and Windows® environment. It lets a DB2 UDB for Linux, UNIX and Windows client recover from a communication failure when the client is connected to a DB2 UDB for Linux, UNIX and Windows database. When a communication failure occurs, DB2 Universal JDBC Driver client reroute support causes the underlying connection to be rerouted to an alternate location where a failover replica of the database resides. When a connection is preserved through a client reroute, an exception is thrown to indicate to the user that a reroute has occurred, and the transaction is rolled back.

DB2 Universal JDBC Driver client reroute support is available only for connections that use the `javax.sql.DataSource` interface.

Connectivity information for the alternate location is provided to Java™ clients by the activeServerListJNDIName property of the primary JDBC `DataSource` instance. activeServerListJNDIName identifies a JNDI reference to a `DB2ActiveServerList` instance in a JNDI repository of alternate server information.

`DB2ActiveServerList` is a serializable Java bean with two properties: `alternateServerName` and `alternatePortNumber`. `getXXX` and `setXXX` methods are defined for each property. The Java bean looks like this:

```
package com.ibm.db2.jcc;
public class DB2ActiveServerList implements java.io.Serializable,
  javax.naming.Referenceable
{
  public String[] alternateServerName;
  public synchronized void
    setAlternateServerName(String[] alternateServer);
  public String[] getAlternateServerName();
  public int[] alternatePortNumber;
  public synchronized void
    setAlternatePortNumber(int[] alternatePortNumberList);
  public int[] getAlternatePortNumber();
}
```

Alternates are propagated from the server to the client dynamically when the client issues a CONNECT or CONNECT RESET. This dynamically propagated alternate server information is stored in global driver memory, and is also updated in the JNDI store of DB2 active servers. The DB2 Universal JDBC Driver attempts to propagate the updated information to the alternate JNDI after failover.

A newly established failover connection is configured with the original `DataSource` properties, except for the server name and port number. In addition, any DB2 special registers that were modified during the original connection are reestablished in the failover connection.

When a communication failure occurs, the DB2 Universal JDBC Driver first attempts recovery to the original server. Reconnection to the original server is called failback. If failback fails, the driver attempts to connect to the alternate location (failover). After a failover or failback connection is reestablished, the driver throws a `java.sql.SQLException` to the application with SQLCODE -4498, to indicate to the application that a failover or failback occurred and the transaction failed. The application can then retry its transaction.

Alternate server setup method: Use JNDI to set up the alternate server. This involves these steps:

1. Set the environment for an initial context. You can do this by creating a jndi.properties file and add its name to the CLASSPATH.

   Example: A jndi.properties file:

   ```
   java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
   java.naming.provider.url=file:/tmp
   ```

2. Create an instance of `DB2ActiveServerList`, and bind that instance to the JNDI registry.

   Example: Code that creates an instance of `DB2ActiveServerList` and binds that instance to the JNDI registry:

   ```
   // Create a starting context for naming operations
   InitialContext registry = new InitialContext();
   // Create a DB2ActiveServerList object
   DB2ActiveServerList address = new DB2ActiveServerList();
   // Set the port number and server name for the alternate server
   int[] a = {50000};
   String[] s = {"mvs3.sj.ibm.com"};
   address.setActivePortNumber(a);
   address.setActiveServerName(s);
   // Bind the DB2ActiveServerList instance to the JNDI registry
   registry.rebind("jdbc/alternate", address);
   ```

3. Assign the logical name of the `DB2ActiveServerList`, object, which contains the alternate server location information, to the activeServerListJNDIName property of the original `DataSource`.

   Example: Code that assigns the logical name of the `DB2ActiveServerList` object to the activeServerListJNDIName property of the a `DataSource` instance named datasource:

   ```
   datasource.setActiveServerListJNDIName("jdbc/alternate");
   ```

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370
- "Summary of DB2 Universal JDBC Driver extensions to JDBC" on page 414

Providing extended client information to the DB2 server with the DB2 Universal JDBC Driver

The DB2 Universal JDBC Driver provides DB2®-only methods that you can use to provide extra information about the client to the server. This information can be used for accounting or workload management. The information is sent to the DB2 server when the application performs an action that accesses the server, such as executing SQL.

The methods are listed in Table 36.

Table 36. Methods that provide client information to the DB2 server

Method	Information provided
setDB2ClientUser	User name for a connection
setDB2ClientWorkstation	Client workstation name for a connection
setDB2ClientApplicationInformation	Name of the application that is working with a connection
setDB2ClientAccountingInformation	Accounting information

To set the extended information:

1. Create a `Connection`.
2. Cast the `java.sql.Connection` object to a `com.ibm.db2.jcc.DB2Connection`.
3. Call any of the methods shown in Table 36 on page 314.
4. Execute an SQL statement to cause the information to be sent to the DB2 server.

The following code performs the previous steps to pass a user name and a workstation name to the DB2 server. The numbers to the right of selected statements correspond to the previously-described steps.

```
public class ClientInfoTest {
  public static void main(String[] args) {
    String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
    try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      String user = "db2adm";
      String password = "db2adm";
      Connection con = DriverManager.getConnection(url,         1
        user, password);
      if (conn instanceof DB2Connection) {
        DB2Connection db2conn = (DB2Connection) conn;           2
        db2conn.setDB2ClientUser("Michael L Thompson");         3
        db2conn.setDB2ClientWorkstation("sjwkstn1");
        // Execute SQL to force extended client information to be sent
        // to the server
        conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
          + "WHERE 0 = 1").executeQuery();                      4       }
    } catch (Throwable e) {
        e.printStackTrace();
      }
  }
}
```

Figure 32. Example of passing extended client information to a DB2 server

Related reference:

- "Summary of DB2 Universal JDBC Driver extensions to JDBC" on page 414

Chapter 16. SQLJ application programming

The sections that follow contain information about writing SQLJ applications.

Basic SQLJ application programming concepts

The topics that follow contain basic information about writing SQLJ applications.

Basic steps in writing an SQLJ application

Writing a SQLJ application has much in common with writing an SQL application in any other language: In general, you need to do the following things:

- Import the Java™ packages that contain SQLJ and JDBC methods.
- Declare variables for sending data to or retrieving data from DB2® tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks, and the order in which you execute those tasks, is somewhat different.

Figure 33 on page 318 is a simple program that demonstrates each task.

```
import sqlj.runtime.*;                                                    1
import java.sql.*;

#sql context EzSqljCtx;                                                   3a
#sql iterator EzSqljNameIter (String LASTNAME);                           4a

public class EzSqlj {
  public static void main(String args[])
    throws SQLException
  {
    EzSqljCtx ctx = null;
    String URLprefix = "jdbc:db2:";
    String url;
    url = new String(URLprefix + args[0]);
                                              // Location name is an input parameter
    String hvmgr="000010";                                                2
    String hvdeptno="A00";
    try {                                                                 3b
      Class.forName("com.ibm.db2.jcc.DB2Driver");
    } catch (Exception e)
    {
       throw new SQLException("Error in EzSqlj: Could not load the driver");
    }
    try
    {
       System.out.println("About to connect using url: " + url);
       Connection con0 = DriverManager.getConnection(url);                3c
                                              // Create a JDBC Connection
       con0.setAutoCommit(false);             // set autocommit OFF
       ctx = new EzSqljCtx(con0);                                         3d

       try
       {
         EzSqljNameIter iter;
         int count=0;

         #sql [ctx] iter =
           {SELECT LASTNAME FROM EMPLOYEE};                               4b
                                              // Create result table of the SELECT
         while (iter.next()) {                                            4c
            System.out.println(iter.LASTNAME()); // Retrieve rows from result table
            count++;
         }
         System.out.println("Retrieved " + count + " rows of data");
       }
```

Figure 33. Simple SQLJ application (Part 1 of 2)

```
      catch( SQLException e )                                              5
      {
         System.out.println ("**** SELECT SQLException...");
         while(e!=null) {
           System.out.println ("Error msg: " + e.getMessage());
           System.out.println ("SQLSTATE: " + e.getSQLState());
           System.out.println ("Error code: " + e.getErrorCode());
           e = e.getNextException(); // Check for chained exceptions
         }
      }
      catch( Exception e )
      {
         System.out.println("**** NON-SQL exception   = " + e);
         e.printStackTrace();
      }
      try
      {
        #sql [ctx]                                                         4d
          {UPDATE DEPARTMENT SET MGRNO=:hvmgr
             WHERE DEPTNO=:hvdeptno};
                                        // Update data for one department  6
        #sql [ctx] {COMMIT};              // Commit the update
      }
      catch( SQLException e )
      {
         System.out.println ("**** UPDATE SQLException...");
         System.out.println ("Error msg: " + e.getMessage() + ".  SQLSTATE=" +
           e.getSQLState() + " Error code=" + e.getErrorCode());
         e.printStackTrace();
      }
      catch( Exception e )
      {
         System.out.println("**** NON-SQL exception   = " + e);
         e.printStackTrace();
      }
      iter.close();                                    // Close the iterator
      ctx.close();                                                         7
  }
  catch(SQLException e)
  {
     System.out.println ("**** SQLException ...");
     System.out.println ("Error msg: " + e.getMessage() + ".  SQLSTATE=" +
       e.getSQLState() + " Error code=" + e.getErrorCode());
     e.printStackTrace();
  }
  catch(Exception e)
  {
    System.out.println ("**** NON-SQL exception = " + e);
    e.printStackTrace();
  }

}
```

Figure 33. Simple SQLJ application (Part 2 of 2)

Notes to Figure 33 on page 318:

1 These statements import the `java.sql` package, which contains the JDBC core API, and the `sqlj.runtime` package, which contains the SQLJ API. For information on other packages or classes that you might need to access, see Access Java packages for SQLJ support.

2 `String` variables hvmgr and hvdeptno are *host identifiers*, which are equivalent to DB2 host variables. See Declare variables in SQLJ applications for more information.

3a, **3b**, **3c**, and **3d** These statements demonstrate how to connect to a data source using one of the three available techniques. See Connect to a data source using SQLJ for more details.

4a , **4b** , **4c** , and **4d**
: These statements demonstrate how to execute SQL statements in SQLJ. Statement 4a demonstrates the SQLJ equivalent of declaring an SQL cursor. Statements 4b and 4c show one way of doing the SQLJ equivalent of executing SQL FETCHes. Statement 4d shows how to do the SQLJ equivalent of performing an SQL UPDATE. For more information, see Execute SQL in an SQLJ application.

5
: This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see Handle errors in an SQLJ application. For more information on handling SQL warnings, see Handle SQL warnings in an SQLJ application.

6
: This is an example of a comment. For rules on including comments in SQLJ programs, see Include comments in an SQLJ application.

7
: This statement closes the connection to the data source. See Close the connection to the data source in an SQLJ application.

Related concepts:

- "Java packages for SQLJ support" on page 320
- "Variables in SQLJ applications" on page 320
- "SQL statements in an SQLJ application" on page 330

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

Java packages for SQLJ support

Before you can execute SQLJ statements or invoke JDBC methods in your SQLJ program, you need to be able to access all or parts of various Java™ packages that contain support for those statements. You can do that either by importing the packages or specific classes, or by using fully-qualified class names. You might need the following packages or classes for your SQLJ program:

sqlj.runtime
: Contains the SQLJ run-time API.

java.sql
: Contains the core JDBC API.

com.ibm.db2.jcc
: Contains the DB2®-specific implementation of JDBC and SQLJ.

javax.naming
: Contains classes and interfaces for Java Naming and Directory Interface (JNDI), which is often used for implementing a `DataSource`.

javax.sql
: Contains JDBC 2.0 standard extensions.

Related concepts:

- "Basic steps in writing an SQLJ application" on page 317

Variables in SQLJ applications

In DB2® programs in other languages, you use host variables to pass data between the application program and DB2. In SQLJ programs, you use *host expressions*. A

host expression can be a simple Java™ identifier, or it can be a complex expression. Every host expression must start with a colon when it is used in an SQL statement. Host expressions are case sensitive.

For the DB2 Universal JDBC Driver, a Java identifier can have any of the data types listed in the Java data type column of Java, JDBC, and SQLJ data types. Data types that are specified in an iterator can be any of the types in the Java data type column of Java, JDBC, and SQLJ data types.

A complex expression is an array element or Java expression that evaluates to a single value. A complex expression in an SQLJ clause must be surrounded by parentheses.

The following examples demonstrate how to use host expressions.

Example: Declaring a Java identifier and using it in a SELECT statement:

In this example, the statement that begins with `#sql` has the same function as a SELECT statement in other languages. This statement assigns the last name of the employee with employee number 000010 to Java identifier `empname`.

```
String empname;
...
#sql [ctxt]
  {SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'};
```

Example: Declaring a Java identifier and using it in a stored procedure call:

In this example, the statement that begins with `#sql` has the same function as an SQL CALL statement in other languages. This statement uses Java identifier `empno` as an input parameter to stored procedure `A`. The value `IN`, which precedes `empno`, specifies that `empno` is an input parameter. The qualifier that indicates how the parameter is used (IN, OUT, or INOUT) must match the corresponding value in the parameter definition that you specified in the CREATE PROCEDURE statement for the stored procedure.

```
String empno = "0000010";
...
#sql [ctxt] {CALL A (:IN empno)};
```

Example: Using a complex expression as a host identifier:

This example uses complex expression (((int)yearsEmployed++/5)*500) as a host expression.

```
#sql [ctxt] {UPDATE EMPLOYEE
  SET BONUS=:(((int)yearsEmployed++/5)*500) WHERE EMPNO=:empID};
```

SQLJ performs the following actions when it processes a complex host expression:

- Evaluates the host expression from left to right before assigning its value to DB2.
- Evaluates side effects, such as operations with postfix operators, according to normal Java rules. All host expressions are fully evaluated before any of their values are passed to DB2.
- Uses Java rules for rounding and truncation.

Therefore, if the value of yearsEmployed is 6 before the UPDATE statement is executed, the value that is assigned to column BONUS by the UPDATE statement is ((int)6/5)*500, or 500. After 500 is assigned to BONUS, the value of yearsEmployed is incremented.

Restrictions on variable names: Two strings have special meanings in SQLJ programs. Observe the following restrictions when you use these strings in your SQLJ programs:

- The string __sJT_ is a reserved prefix for variable names that are generated by SQLJ. Do not begin the following types of names with __sJT_:
  - Host expression names
  - Java variable names that are declared in blocks that include executable SQL statements
  - Names of parameters for methods that contain executable SQL statements
  - Names of fields in classes that contain executable SQL statements, or in classes with subclasses or enclosed classes that contain executable SQL statements
- The string _SJ is a reserved suffix for resource files and classes that are generated by SQLJ. Avoid using the string _SJ in class names and input source file names.

Related concepts:

- "Basic steps in writing an SQLJ application" on page 317

Related reference:

- "Java, JDBC, and SQL data types" on page 365

Comments in an SQLJ application

To document your program, you need to include comments. To do that, use Java™ comments. Java comments are denoted by `/* */` or `//`. You can include Java comments outside SQLJ clauses, wherever the Java language permits them. Within an SQLJ clause, you can use Java comments only within host expressions.

Related concepts:

- "Basic steps in writing a JDBC application" on page 263

Connecting to a data source using SQLJ

In an SQLJ application, as in any other DB2® application, you must be connected to a database server before you can execute SQL statements. In SQLJ, as in JDBC, a database server is called a *data source*.

You can use one of five techniques to connect to a data source:

- Explicitly create a connection using the JDBC `DriverManager` interface. There are two techniques for doing this.
- Explicitly create a connection using the JDBC `DataSource` interface. There are two techniques for doing this.
- Implicitly create a connection.

Connection technique 1: This technique uses the JDBC `DriverManager` as the underlying means for creating the connection. Use it with any level of the JDBC driver.

1. Execute an SQLJ *connection declaration clause*.

   Doing this generates a *connection context class*. The simplest form of the connection declaration clause is:

   ```
   #sql context context-class-name;
   ```

The name of the generated connection context class is *context-class-name*.

2. Load a JDBC driver by invoking the `Class.forName` method:
   - For the DB2 Universal JDBC Driver, invoke `Class.forName` this way:
     ```
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```
   - For the DB2 JDBC Type 2 Driver, invoke `Class.forName` this way:
     ```
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
     ```
3. Invoke the constructor for the connection context class that you created in step 1 on page 322.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

   ```
   connection-context-class connection-context-object=
     new connection-context-class(String url, boolean autocommit);

   connection-context-class connection-context-object=
     new connection-context-class(String url, String user,
       String password, boolean autocommit);
   connection-context-class connection-context-object=
     new connection-context-class(String url, Properties info,
       boolean autocommit);
   ```

   The meanings of the parameters are:

   url A string that specifies the location name that is associated with the data source. That argument has one of the forms that are specified in Connect to a data source using the DriverManager interface with the JDBC Universal Driver. The form depends on which JDBC driver you are using.

   user **and** *password*
   Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

   info
   Specifies an object of type `java.util.Properties` that contains a set of driver properties for the connection. For the DB2 JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver), you should specify only the `user` and `password` properties. For the DB2 Universal JDBC Driver, you can specify any of the properties listed in Properties for the DB2 Universal JDBC Driver.

   autocommit
   Specifies whether you want the database manager to issue a COMMIT after every statement. Possible values are `true` or `false`. If you specify `false`, you need to do explicit commit operations.

The following code uses connection technique 1 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql context Ctx;            // Create connection context class Ctx        1
String userid="dbadm";       // Declare variables for user ID and password
String password="dbadm";
String empname;              // Declare a host variable
...
try {                        // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");                              2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Ctx myConnCtx=                                                             3
  new Ctx("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
  userid,password,false);    // Create connection context object myConnCtx
                             // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                             // Use myConnCtx for executing an SQL statement
```

Figure 34. Using connection technique 1 to connect to a data source

Connection technique 2: This technique uses the JDBC `DriverManager` interface for creating the connection. Use it with any level of the JDBC driver.

1. Execute an SQLJ connection declaration clause.

   This is the same as step 1 on page 322 in connection technique 1.

2. Load the driver.

   This is the same as step 2 on page 323 in connection technique 1.

3. Invoke the JDBC `DriverManager.getConnection` method.

   Doing this creates a JDBC connection object for the connection to the data source. You can use any of the forms of `getConnection` that are specified in Connect to a data source using the DriverManager interface with the JDBC Universal Driver.

   The meanings of the *url*, *user*, and *password* parameters are the same as the meanings of the parameters in step 3 on page 323 of connection technique 1.

4. Invoke the constructor for the connection context class that you created in step 1.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in the following form:

   ```
   connection-context-class connection-context-object=
     new connection-context-class(Connection JDBC-connection-object);
   ```

   The *JDBC-connection-object* parameter is the `Connection` object that you created in step 3.

The following code uses connection technique 2 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql context Ctx;           // Create connection context class Ctx        1
String userid="dbadm";      // Declare variables for user ID and password
String password="dbadm";
String empname;             // Declare a host variable
...
try {                       // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");                            2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=                                                      3
  DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
    userid,password);
                            // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit                       4
Ctx myConnCtx=new Ctx(jdbccon);                                          5
                            // Create connection context object myConnCtx
                            // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                            // Use myConnCtx for executing an SQL statement
```

Figure 35. Using connection technique 2 to connect to a data source

Connection technique 3: This technique uses the JDBC `DataSource` interface for creating the connection.

1. Execute an SQLJ connection declaration clause.

   This is the same as step 1 on page 322 in connection technique 1.

2. If your system administrator created a `DataSource` object in a different program:

   a. Obtain the logical name of the data source to which you need to connect.

   b. Create a context to use in the next step.

   c. In your application program, use the Java™ Naming and Directory Interface (JNDI) to get the `DataSource` object that is associated with the logical data source name.

   Otherwise, create a `DataSource` object and assign properties to it, as shown in "Creating and using a data source in the same application" in Connect to a data source using the DataSource interface.

3. Invoke the JDBC `DataSource.getConnection` method.

   Doing this creates a JDBC connection object for the connection to the data source. You can one of the following forms of `getConnection`:

   ```
   getConnection();
   getConnection(user, password);
   ```

   The meanings of *user* and *password* parameters are the same as the meanings of the parameters in step 3 on page 323 of connection technique 1.

4. If the default autocommit mode is not appropriate, invoke the JDBC `Connection.setAutoCommit` method.

   Doing this indicates whether you want the database manager to issue a COMMIT after every statement. The form of this method is:

   ```
   setAutoCommit(boolean autocommit);
   ```

5. Invoke the constructor for the connection context class that you created in step 1.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in the following form:

```
connection-context-class connection-context-object=
  new connection-context-class(Connection JDBC-connection-object);
```

The *JDBC-connection-object* parameter is the `Connection` object that you created in step 3 on page 325.

The following code uses connection technique 3 to create a connection to a location with logical name `jdbc/sampledb`. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
#sql context CtxSqlj;       // Create connection context class CtxSqlj   1
Context ctx=new InitialContext();                                         2b
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");                    2c
Connection con=ds.getConnection();                                        3
String empname;             // Declare a host variable
...
con.setAutoCommit(false);   // Do not autocommit                          4
CtxSqlj myConnCtx=new CtxSqlj(con);                                       5
                            // Create connection context object myConnCtx
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                            // Use myConnCtx for executing an SQL statement
```

Figure 36. Using connection technique 3 to connect to a data source

Connection technique 4 (DB2 Universal JDBC Driver only): This technique uses the JDBC `DataSource` interface for creating the connection. This technique **requires** that the `DataSource` is registered with JNDI.

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Execute an SQLJ connection declaration clause.

   For this type of connection, the connection declaration clause needs to be of this form:

   ```
   #sql public static context context-class-name
    with (dataSource="logical-name");
   ```

   The connection context must be declared as public and static. *logical-name* is the data source name that you obtained in step 1.
3. Invoke the constructor for the connection context class that you created in step 2.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

   ```
   connection-context-class connection-context-object=
     new connection-context-class();

   connection-context-class connection-context-object=
     new connection-context-class (String user,
       String password);
   ```

   The meanings of the *user* and *password* parameters are the same as the meanings of the parameters in step 3 on page 323 of connection technique 1.

The following code uses connection technique 4 to create a connection to a location with logical name jdbc/sampledb. The connection requires a user ID and password.

```
#sql public static context Ctx
  with (dataSource="jdbc/sampledb");                                    2
                                // Create connection context class Ctx
String userid="dbadm";          // Declare variables for user ID and password
String password="dbadm";

String empname;                 // Declare a host variable
...
Ctx myConnCtx=new Ctx(userid, password);                                3
                                // Create connection context object myConnCtx
                                // for the connection to jdbc/sampledb
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                                // Use myConnCtx for executing an SQL statement
```

Figure 37. Using connection technique 4 to connect to a data source

Connection technique 5: This technique uses the default connection to connect to the data source. You use the default connection by specifying your SQL statements without a connection context object. When you use this technique, you do not need to load a JDBC driver unless you explicitly use JDBC interfaces in your program. For example:

```
#sql {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};  // Use default connection for
                                    // executing an SQL statement
```

To create a default connection context, SQLJ does a JNDI lookup for jdbc/defaultDataSource. If nothing is registered, a null context exception is issued when SQLJ attempts to access the context.

Related concepts:

- "How JDBC applications connect to a data source" on page 267

Related tasks:

- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270
- "Connecting to a data source using the DataSource interface" on page 272

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Setting the isolation level for an SQLJ transaction

To set the isolation level for a unit of work within an SQLJ program, use the SET TRANSACTION ISOLATION LEVEL clause. Table 37 shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their DB2® equivalents.

Table 37. Equivalent SQLJ and DB2 isolation levels

SET TRANSACTION value	DB2 isolation level
SERIALIZABLE	Repeatable read
REPEATABLE READ	Read stability

Table 37. Equivalent SQLJ and DB2 isolation levels (continued)

SET TRANSACTION value	DB2 isolation level
READ COMMITTED	Cursor stability
READ UNCOMMITTED	Uncommitted read

The isolation level affects the underlying JDBC connection as well as the SQLJ connection. You can change the isolation level only at the beginning of a transaction.

Related concepts:

- "Isolation levels" in the *SQL Reference, Volume 1*

Committing or rolling back SQLJ transactions

If you disable autocommit for an SQLJ connection, you need to perform explicit commit or rollback operations. You do this using execution clauses that contain the SQL COMMIT or ROLLBACK statements:

```
#sql [myConnCtx] {COMMIT};
#sql [myConnCtx] {ROLLBACK};
```

Related concepts:

- "Savepoints in SQLJ applications" on page 328

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

Savepoints in SQLJ applications

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

Under the DB2 Universal JDBC Driver, you can include any form of the SQL SAVEPOINT statement in your SQLJ program.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```
#sql context Ctx;             // Create connection context class Ctx
String empNumVar;
int shoeSizeVar;
...
try {                         // Load the JDBC driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=
  DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
    userid,password);
                              // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx ctxt=new Ctx(jdbccon);
                              // Create connection context object myConnCtx
                              // for the connection to NEWYORK
#sql [ctxt] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};
                              // Create a distinct type
#sql [ctxt] {COMMIT};
                              // Commit the create
#sql [ctxt]
  {CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)};
                              // Create table with distinct type
#sql [ctxt] {COMMIT};
                              // Commit the create
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000010', 6)};
                              // Insert a row
#sql [ctxt]
  {SAVEPOINT SVPT1 ON ROLLBACK RETAIN CURSORS};
                              // Create a savepoint
...
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000020', 10)};
                               // Insert another row
#sql [ctxt] {ROLLBACK TO SAVEPOINT SVPT1};
                               // Roll back work to the point
                               //  after the first insert
...
#sql [ctxt] {RELEASE SAVEPOINT SVPT1};
                               // Release the savepoint
ctx.close();                   // Close the connection context
```

Figure 38. Setting, rolling back to, and releasing a savepoint in an SQLJ application

Related tasks:

- "Committing or rolling back SQLJ transactions" on page 328

Related reference:

- "ROLLBACK statement" in the *SQL Reference, Volume 2*
- "RELEASE SAVEPOINT statement" in the *SQL Reference, Volume 2*
- "SAVEPOINT statement" in the *SQL Reference, Volume 2*

Closing the connection to a data source in an SQLJ application

When you have finished with a connection to a data source, you need to close the connection to the data source. Doing so releases the connection context object's DB2® and SQLJ resources immediately.

To close the connection to the data source, use the `ConnectionContext.close()` method. This closes the connection context, as well as the connection to the data source. For example:

```
...
ctx = new EzSqljctx(con0);          // Create a connection context object
                                    // from JDBC connection con0
...                                 // Perform various SQL operations
 EzSqljctx.close();                 // Close the connection context and
                                    // connection to the data source
```

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

SQL statements in an SQLJ application

You execute SQL statements in a traditional SQL program to create tables, insert, update, and delete data in tables, retrieve data from the tables, call stored procedures, or commit or roll back transactions. In an SQLJ program, you also execute these statements, within SQLJ *executable clauses*. An executable clause can have one of the following general forms:

```
#sql [connection-context] {sql-statement};
#sql [connection-context,execution-context] {sql-statement};
#sql [execution-context] {sql-statement};
```

In an executable clause, you should *always* specify an explicit connection context, with one exception: you do not specify an explicit connection context for a FETCH statement. You include an execution context only for specific cases. See Control the execution of SQL statements in SQLJ for information about when you need an execution context.

Related concepts:

- "Comments in an SQLJ application" on page 322
- "Using SQLJ and JDBC in the same application" on page 345
- "LOBs in SQLJ applications with the DB2 Universal JDBC Driver" on page 348
- "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 354
- "How an SQLJ application retrieves data from DB2 tables" on page 331

Related tasks:

- "Making batch updates in SQLJ applications" on page 355
- "Calling stored procedures in an SQLJ application" on page 343
- "Committing or rolling back SQLJ transactions" on page 328
- "Creating and modifying DB2 objects in an SQLJ application" on page 331
- "Handling SQL errors in an SQLJ application" on page 343
- "Setting the isolation level for an SQLJ transaction" on page 327
- "Using a named iterator in an SQLJ application" on page 332
- "Using a positioned iterator in an SQLJ application" on page 334
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336
- "Using scrollable iterators in an SQLJ application" on page 361
- "Handling SQL warnings in an SQLJ application" on page 344
- "Controlling the execution of SQL statements in SQLJ" on page 353

Related reference:

- "SQLJ executable-clause" on page 401

Creating and modifying DB2 objects in an SQLJ application

Use SQLJ executable clauses to do the following things:

- Execute data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE)
- Execute INSERT, searched UPDATE, and searched DELETE statements

For example, the following executable statements demonstrate an INSERT, a searched UPDATE, and a searched DELETE:

```
#sql [myConnCtx] {INSERT INTO DEPARTMENT VALUES
  ("X00","Operations 2","000030","E01",NULL)};
#sql [myConnCtx] {UPDATE DEPARTMENT
  SET MGRNO="000090" WHERE MGRNO="000030"};
#sql [myConnCtx] {DELETE FROM DEPARTMENT
  WHERE DEPTNO="X00"};
```

For information on positioned UPDATEs and DELETEs, see Perform positioned UPDATE and DELETE operations in an SQLJ application.

Related tasks:

- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336

How an SQLJ application retrieves data from DB2 tables

Just as in DB2® applications in other languages, if you want to retrieve a single row from a DB2 table in an SQLJ application, you can write a SELECT INTO statement with a WHERE clause that defines a result table that contains only that row:

```
#sql [myConnCtx] {SELECT DEPTNO INTO :hvdeptno
  FROM DEPARTMENT WHERE DEPTNAME="OPERATIONS"};
```

However, most SELECT statements that you use create result tables that contain many rows. In DB2 applications in other languages, you use a cursor to select the individual rows from the result table. That cursor can be non-scrollable, which means that when you use it to fetch rows, you move the cursor serially, from the beginning of the result table to the end. Alternatively, the cursor can be scrollable, which means that when you use it to fetch rows, you can move the cursor forward, backward, or to any row in the result table.

The SQLJ equivalent of a cursor is a *result set iterator*. Like a cursor, a result set iterator can be non-scrollable or scrollable. This topic discusses how to use non-scrollable iterators. For information on using scrollable iterators, see Use scrollable iterators in an SQLJ application.

A result set iterator is a Java™ object that you use to retrieve rows from a result table. Unlike a cursor, a result set iterator can be passed as a parameter to a method.

The basic steps in using a result set iterator are:

1. Declare the iterator, which results in an iterator class
2. Define an instance of the iterator class.
3. Assign the result table of a SELECT to an instance of the iterator.
4. Retrieve rows.

5. Close the iterator.

There are two types of iterators: *positioned iterators* and *named iterators*. Postitioned iterators extend the interface `sqlj.runtime.PositionedIterator`. Positioned iterators identify the columns of a result table by their position in the result table. Named iterators extend the interface `sqlj.runtime.NamedIterator`. Named iterators identify the columns of the result table by result table column names.

Related tasks:

- "Using a named iterator in an SQLJ application" on page 332
- "Using a positioned iterator in an SQLJ application" on page 334
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336

Related reference:

- "SQLJ iterator-declaration-clause" on page 400

Using a named iterator in an SQLJ application

The steps in using a named iterator are:

1. Declare the iterator.

   You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name as the iterator. For a named iterator, the iterator declaration clause specifies the following information:

   - The name of the iterator
   - A list of column names and Java™ data types
   - Information for a Java class declaration, such as whether the iterator is `public` or `static`
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   When you declare a named iterator for a query, you specify names for each of the iterator columns. Those names must match the names of columns in the result table for the query. An iterator column name and a result table column name that differ only in case are considered to be matching names. The named iterator class that results from the iterator declaration clause contains *accessor methods*. There is one accessor method for each column of the iterator. Each accessor method name is the same as the corresponding iterator column name. You use the accessor methods to retrieve data from columns of the result table.

   You need to specify Java data types in the iterators that closely match the corresponding DB2® column data types. See Java, JDBC, and SQL data types for a list of the best mappings between Java data types and DB2 data types.

   You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as `public`. Therefore, if an iterator needs to be `public`, it can be declared only where a `public` class is allowed. The following list describes some alternative methods of declaring an iterator:

   - As `public`, in a source file by itself

This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or `public` classes in the same source file.

- As a top-level class in a source file that contains other top-level class definitions

  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.

- As a nested static class within another class

  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible to other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.

- As an inner class within another class

  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.

  You cannot cast a JDBC `ResultSet` to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See Use SQLJ and JDBC in the same application for more information on casting a `ResultSet` to a iterator.

2. Create an instance of the iterator class.

   You declare an object of the named iterator class to retrieve rows from a result table.

3. Assign the result table of a SELECT to an instance of the iterator.

   To assign the result table of a SELECT to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a named iterator is:

   ```
   #sql context-clause iterator-object={select-statement};
   ```

   See SQLJ assignment-clause and SQLJ context-clause for more information.

4. Retrieve rows.

   Do this by invoking accessor methods in a loop. Accessor methods have the same names as the corresponding columns in the iterator, and have no parameters. An accessor method returns the value from the corresponding column of the current row in the result table. Use the `NamedIterator.next()` method to move the cursor forward through the result table.

   To test whether you have retrieved all rows, check the value that is returned when you invoke the `next` method. `next` returns a `boolean` with a value of `false` if there is no next row.

5. Close the iterator.

   Use the `NamedIterator.close` method to do this.

The following code demonstrates how to declare and use a named iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql  iterator ByName(String LastName, Date HireDate);                    1
                                // Declare named iterator ByName
{
  ByName nameiter;              // Declare object of ByName class          2
  #sql [ctxt]
 nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                       3
                                // Assign the result table of the SELECT
                                // to iterator object nameiter
  while (nameiter.next())       // Move the iterator through the result    4
                                // table and test whether all rows retrieved
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());   // Use accessor methods LastName and
                                // HireDate to retrieve column values
  }
  nameiter.close();             // Close the iterator                      5
}
```

Figure 39. Using a named iterator

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 345

Related tasks:

- "Using a positioned iterator in an SQLJ application" on page 334
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336

Related reference:

- "Java, JDBC, and SQL data types" on page 365
- "SQLJ assignment-clause" on page 405
- "SQLJ context-clause" on page 402

Using a positioned iterator in an SQLJ application

The steps in using a positioned iterator are:

1. Declare the iterator.

   You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name and attributes as the iterator. For a positioned iterator, the iterator declaration clause specifies the following information:

   - The name of the iterator
   - A list of Java™ data types
   - Information for a Java class declaration, such as whether the iterator is `public` or `static`
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   The data type declarations represent columns in the result table and are referred to as columns of the result set iterator. The columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table.

You need to specify Java data types in the iterators that closely match the corresponding DB2® column data types. SeeJava, JDBC, and SQL data types for a list of the best mappings between Java data types and DB2 data types.

You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as `public`. Therefore, if an iterator needs to be `public`, it can be declared only where a `public` class is allowed. The following list describes some alternative methods of declaring an iterator:

- As `public`, in a source file by itself

  This is the most versatile method of declaring an iterator. This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or `public` classes in the same source file.
- As a top-level class in a source file that contains other top-level class definitions

  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.
- As a nested static class within another class

  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible from other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.
- As an inner class within another class

  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.

  You cannot cast a JDBC `ResultSet` to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See Use SQLJ and JDBC in the same application for more information on casting a `ResultSet` to a iterator.

2. Create an instance of the iterator class.

   You declare an object of the positioned iterator class to retrieve rows from a result table.
3. Assign the result table of a SELECT to an instance of the iterator.

   To assign the result table of a SELECT to an iterator, you use an SQLJ *assignment clause*. The format of the assignment clause for a positioned iterator is:

   ```
   #sql context-clause iterator-object={select-statement};
   ```
4. Retrieve rows.

   Do this by executing FETCH statements in executable clauses in a loop. The FETCH statements looks the same as a FETCH statements in other languages.

   To test whether you have retrieved all rows, invoke the `PositionedIterator.endFetch` method after each FETCH. `endFetch` returns a `boolean` with the value `true` if the FETCH failed because there are no rows to retrieve.
5. Close the iterator.

Use the `PositionedIterator.close` method to do this.

The following code demonstrates how to declare and use a positioned iterator. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator ByPos(String,Date); // Declare positioned iterator ByPos  1
{
  ByPos positer;                  // Declare object of ByPos class    2
  String name = null;             // Declare host variables
  Date hrdate;
  #sql [ctxt] positer =
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                         3
                                  // Assign the result table of the SELECT
                                  // to iterator object positer
  #sql {FETCH :positer INTO :name, :hrdate };                         4
                                  // Retrieve the first row
  while (!positer.endFetch())     // Check whether the FETCH returned a row
  { System.out.println(name + " was hired in " +
      hrdate);
    #sql {FETCH :positer INTO :name, :hrdate };
                                  // Fetch the next row
  }
  positer.close();                // Close the iterator               5
}
```

Figure 40. Using a positioned iterator

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 345
- "How an SQLJ application retrieves data from DB2 tables" on page 331

Related tasks:

- "Using a named iterator in an SQLJ application" on page 332

Related reference:

- "Java, JDBC, and SQL data types" on page 365

Performing positioned UPDATE and DELETE operations in an SQLJ application

As in DB2® applications in other languages, performing positioned UPDATEs and DELETEs is an extension of retrieving rows from a result table. The basic steps are:

1. Declare the iterator.

   The iterator can be positioned or named. For positioned UPDATE or DELETE operations, the iterator must be declared as updatable. To do this, the declaration must include the following clauses:

   implements sqlj.runtime.ForUpdate
   This clause causes the generated iterator class to include methods for using updatable iterators. This clause is required for programs with positioned UPDATE or DELETE operations.

   with (updateColumns=*"column-list"***)**
   This clause specifies a comma-separated list of the columns of the result table that the iterator will update. This clause is optional.

You need to declare the iterator as `public`, so you need to follow the for declaring and using `public` iterators in the same file or different files.

If you declare the iterator in a file by itself, any SQLJ source file that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator. The authorization ID under which a positioned UPDATE or DELETE statement executes depends on whether the statement executes statically or dynamically. If the statement executes statically, the authorization ID is the owner of the DB2 plan or package that includes the statement. If the statement executes dynamically the authorization ID is determined by the DYNAMICRULES behavior that is in effect. For the DB2 Universal JDBC Driver, the behavior is always DYNAMICRULES BIND.

2. Disable autocommit mode for the connection.

   If autocommit mode is enabled, a COMMIT operation occurs every time the positioned UPDATE statement executes, which causes the iterator to be destroyed unless the iterator has the `with (holdability=true)` attribute. Therefore, you need to turn autocommit off to prevent COMMIT operations until you have finished using the iterator. If you want a COMMIT to occur after every update operation, an alternative way to keep the iterator from being destroyed after each COMMIT operation is to declare the iterator `with (holdability=true)`.

3. Create an instance of the iterator class.

   This is the same step as for a non-updatable iterator.

4. Assign the result table of a SELECT to an instance of the iterator.

   This is the same step as for a non-updatable iterator. The SELECT statement must not include a FOR UPDATE clause.

5. Retrieve and update rows.

   For a positioned iterator, do this by performing the following actions in a loop:

   a. Execute a FETCH statement in an executable clause to obtain the current row.
   b. Test whether the iterator is pointing to a row of the result table by invoking the `PositionedIterator.endFetch` method.
   c. If the iterator is pointing to a row of the result table, execute an SQL UPDATE... WHERE CURRENT OF *:iterator-object* statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF *:iterator-object* statement in an executable clause to delete the current row.

   For a named iterator, do this by performing the following actions in a loop:

   a. Invoke the `next` method to move the iterator forward.
   b. Test whether the iterator is pointing to a row of the result table by checking whether `next` returns `true`.
   c. Execute an SQL UPDATE... WHERE CURRENT OF *iterator-object* statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF *iterator-object* statement in an executable clause to delete the current row.

6. Close the iterator.

   Use the `close` method to do this.

The following code shows how to declare a positioned iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.

First, in one file, declare positioned iterator UpdByPos, specifying that you want to use the iterator to update column SALARY:

```
import java.math.*;    // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate       1
  with(updateColumns="SALARY") (String, BigDecimal);
```

Figure 41. Declaring a positioned iterator for a positioned UPDATE

Then, in another file, use UpdByPos for a positioned UPDATE, as shown in the following code fragment:

```
import sqlj.runtime.*;       // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByPos;             // Import the generated iterator class that
                             // was created by the iterator declaration clause
                             // for UpdByName in another file
#sql context HSCtx;          // Create a connnection context class HSCtx
public static void main (String args[])
{
  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch (ClassNotFoundException e) {
    e.printStackTrace();
          }
  Connection HSjdbccon=
  DriverManager.getConnection("jdbc:db2:SANJOSE");
                     // Create a JDBC connection object
  HSjdbccon.setAutoCommit(false);
                     // Set autocommit off so automatic commits       2
                     // do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
                     // Create a connection context object
  UpdByPos upditer;  // Declare iterator object of UpdByPos class     3
  String enum;       // Declares host variable to receive EMPNO
  BigDecimal sal;    // and SALARY column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                     4
      WHERE WORKDEPT='D11'};
                     // Assign result table to iterator object
  #sql {FETCH :upditer INTO :enum,:sal};                              5a
                             // Move cursor to next row
  while (!upditer.endFetch())                                         5b
                             // Check if on a row
  {
    #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
      WHERE CURRENT OF :upditer};                                     5c
                             // Perform positioned update
    System.out.println("Updating row for " + enum);
    #sql {FETCH :upditer INTO :enum,:sal};
                             // Move cursor to next row
  }
  upditer.close();           // Close the iterator                   6
  #sql [myConnCtx] {COMMIT};
                             // Commit the changes
  myConnCtx.close();         // Close the connection context
}
```

Figure 42. Performing a positioned UPDATE with a positioned iterator

The following code shows how to declare a named iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.

First, in one file, declare named iterator `UpdByName`, specifying that you want to use the iterator to update column `SALARY`:

```
import java.math.*;          // Import this class for BigDecimal data type
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate     1
  with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

Figure 43. Declaring a named iterator for a positioned UPDATE

Then, in another file, use `UpdByName` for a positioned UPDATE, as shown in the following code fragment:

```
import sqlj.runtime.*;       // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByName;            // Import the generated iterator class that
                             // was created by the iterator declaration clause
                             // for UpdByName in another file
#sql context HSCtx;          // Create a connnection context class HSCtx
public static void main (String args[])
{
  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch (ClassNotFoundException e) {
    e.printStackTrace();
          }
  Connection HSjdbccon=
  DriverManager.getConnection("jdbc:db2:SANJOSE");
                         // Create a JDBC connection object
  HSjdbccon.setAutoCommit(false);
                         // Set autocommit off so automatic commits  2
                         // do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
                         // Create a connection context object
  UpdByName upditer;                                                  3
                         // Declare iterator object of UpdByName class
  String enum;               // Declare host variable to receive EmpNo
                             // column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                     4
      WHERE WORKDEPT='D11'};
                             // Assign result table to iterator object
  while (upditer.next())                                              5a, 5b
                             // Move cursor to next row and
                             // check ifon a row
  {
    enum = upditer.EmpNo();  // Get employee number from current row
    #sql [myConnCtx]
      {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
        WHERE CURRENT OF :upditer};                                   5c
                             // Perform positioned update
    System.out.println("Updating row for " + enum);
  }
  upditer.close();           // Close the iterator                   6
  #sql [myConnCtx] {COMMIT};
                             // Commit the changes
  myConnCtx.close();         // Close the connection context
}
```

Figure 44. Performing a positioned UPDATE with a named iterator

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 331
- "Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 359

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

Multiple open iterators for the same SQL statement in an SQLJ application

If you are using the DB2 Universal JDBC Driver, and your application connects to a DB2 UDB for z/OS® Version 8 server, or a DB2 UDB for Linux, UNIX®, and Windows® server at the FixPak 4 level or later, you can have multiple concurrently open iterators for a single SQL statement in an SQLJ application. With this capability, you can perform one operation on a table using one iterator while you perform a different operation on the same table using another iterator.

When you use concurrently open iterators in an application, you should close iterators when you no longer need them to prevent excessive storage consumption in the Java™ heap.

The following examples demonstrate how to perform the same operations on a table without concurrently open iterators on a single SQL statement and with concurrently open iterators on a single SQL statement. These examples use the following iterator declaration:

```
import java.math.*;
#sql public iterator  MultiIter(String EmpNo, BigDecimal Salary);
```

Without the capability for multiple, concurrently open iterators for a single SQL statement, if you want to select employee and salary values for a specific employee number, you need to define a different SQL statement for each employee number, as shown in Figure 45.

```
MultiIter iter1 = null;            // Iterator instance for retrieving
                                   // data for first employee
String EmpNo1 = "000100";          // Employee number for first employee
#sql [ctx] iter2 =
  {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo1};
                                   // Assign result table to first iterator
MultiIter iter2 = null;            // Iterator instance for retrieving
                                   // data for second employee
String EmpNo2 = "000200";          // Employee number for second employee
#sql [ctx] iter2 =
  {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo2};
                                   // Assign result table to second iterator
// Process with iter1
// Process with iter2
iter1.close();                     // Close the iterators
iter2.close();
```

Figure 45. Example of concurrent table operations using iterators with different SQL statements

Figure 46 on page 342 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single SQL statement.

```
...
MultiIter iter1 = openIter("000100"); // Invoke openIter to assign the result table
                                      // (for employee 100) to the first iterator
MultiIter iter2 = openIter("000200"); // Invoke openIter to assign the result
                                      // table to the second iterator
                                      // iter1 stays open when iter2 is opened
// Process with iter1
// Process with iter2
...
iter1.close();                        // Close the iterators
iter2.close();
...
public MultiIter openIter(String EmpNo)
                                      // Method to assign a result table
                                      // to an iterator instance
{
  MultiIter iter;
  #sql [ctxt] iter =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
  return iter;                        // Method returns an iterator instance
}
```

Figure 46. Example of concurrent table operations using iterators with the same SQL statement

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 331

Multiple open instances of an iterator in an SQLJ application

Multiple instances of an iterator can be open concurrently in a single SQLJ application. One application for this ability is to open several instances of an iterator that uses host expressions. Each instance can use a different set of host expression values.

The following example shows an application with two concurrently open instances of an iterator.

```
...
ResultSet myFunc(String empid)  // Method to open an iterator and get a resultSet
{
  MyIter iter;
  #sql iter = {SELECT * FROM EMPLOYEE WHERE EMPNO = :empid};
  return iter.getResultSet();
}

// An application can call this method to get a resultSet for each
// employee ID. The application can process each resultSet separately.
...
ResultSet rs1 = myFunc("000100");  // Get employee record for employee ID 000100
...
ResultSet rs2 = myFunc("000200");  // Get employee record for employee ID 000200
```

Figure 47. Example of opening more than one instance of an iterator in a single application

As with any other iterator, you need to remember to close this iterator after the last time you use it to prevent excessive storage consumption.

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 331

Calling stored procedures in an SQLJ application

To call a stored procedure, you use an executable clause that contains an SQL CALL statement. You can execute the CALL statement with host identifier parameters. The basic steps in calling a stored procedure are:

1. Assign values to input (IN or INOUT) parameters.
2. Call the stored procedure.
3. Process output (OUT or INOUT) parameters.
4. If the stored procedure returns multiple result sets, retrieve those result sets. See Retrieve multiple result sets from a stored procedure in an SQLJ application.

The following code illustrates calling a stored procedure that has three input parameters and three output parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```
String FirstName="TOM";            // Input parameters                 1
String LastName="NARISINST";
String Address="IBM";
int CustNo;                        // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName,                      2
                           :IN LastName,
                           :IN Address,
                           :OUT CustNo,
                           :OUT Mark,
                           :OUT MarkErrorText)};
                                 // Call the stored procedure
System.out.println("Output parameters from ADD_CUSTOMER call: ");
System.out.println("Customer number for " + LastName + ": " + CustNo);  3
System.out.println(Mark);
If (MarkErrorText != null)
  System.out.println(" Error messages:" + MarkErrorText);
```

Figure 48. Calling a stored procedure in an SQLJ application

Related concepts:

- "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 354

Handling SQL errors in an SQLJ application

SQLJ clauses use the JDBC class `java.sql.SQLException` for error handling. SQLJ generates an `SQLException` under the following circumstances:

- When any SQL statement returns a negative SQL error code
- When a SELECT INTO SQL statement returns a +100 SQL error code

You can use the `getErrorCode` method to retrieve SQL error codes and the `getSQLState` method to retrieve SQLSTATEs.

To handle SQL errors in your SQLJ application, import the `java.sql.SQLException` class, and use the Java™ error handling `try/catch` blocks to modify program flow when an SQL error occurs. For example:

```
try {
  #sql [ctxt] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e) {
  System.out.println("Error code returned: " + e.getErrorCode());
}
```

With the DB2 Universal JDBC Driver, you can retrieve the SQLCA. For information on writing code to retrieve the SQLCA with the DB2 Universal JDBC Driver, see Handle an SQLException under the DB2 Universal JDBC Driver.

For the DB2 JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver), use the standard `SQLException` to retrieve SQL error information.

Related tasks:

- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282

Handling SQL warnings in an SQLJ application

Other than a +100 SQL error code on a SELECT INTO statement, DB2® warnings do not throw SQLExceptions. To handle DB2 warnings, you need to give the program access to the `java.sql.SQLWarning` class. If you want to retrieve DB2-specific information about a warning, you also need to give the program access to the `com.ibm.db2.jcc.DB2Diagnosable` interface and the `com.ibm.db2.jcc.DB2Sqlca` class. To check for a DB2 warning, invoke the `getWarnings` method after you execute an SQLJ clause. `getWarnings` returns the first `SQLWarning` object that an SQL statement generates. Subsequent `SQLWarning` objects are chained to the first one.

To retrieve DB2-specific information from the `SQLWarning` object with the DB2 Universal JDBC Driver, follow the instructions in Handle an SQLException under the DB2 Universal JDBC Driver.

Before you can execute `getWarnings` for an SQL clause, you need to set up an execution context for that SQL clause. See Control the execution of SQL statements in SQLJ for information on how to set up an execution context. The following example demonstrates how to retrieve an `SQLWarning` object for an SQL clause with execution context execCtx:

```
ExecutionContext execCtx=myConnCtx.getExecutionContext();
                                    // Get default execution context from
                                    // connection context
SQLWarning sqlWarn;
...
#sql [myConnCtx,execCtx] {SELECT LASTNAME INTO :empname
  FROM EMPLOYEE WHERE EMPNO='000010'};
if ((sqlWarn = execCtx.getWarnings()) != null)
System.out.println("SQLWarning " + sqlWarn);
```

Related tasks:

- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Controlling the execution of SQL statements in SQLJ" on page 353
- "Handling SQL errors in an SQLJ application" on page 343

Advanced SQLJ application programming concepts

The topics that follow contain more advanced information about writing SQLJ applications.

Using SQLJ and JDBC in the same application

You can combine SQLJ clauses and JDBC calls in a single program. To do this effectively, you need to be able to do the following things:

- Use a JDBC `Connection` to build an SQLJ `ConnectionContext`, or obtain a JDBC `Connection` from an SQLJ `ConnectionContext`.
- Use an SQLJ iterator to retrieve data from a JDBC `ResultSet` or generate a JDBC `ResultSet` from an SQLJ iterator.

Building an SQLJ ConnectionContext from a JDBC Connection: To do that:

1. Execute an SQLJ connection declaration clause to create a `ConnectionContext` class.
2. Load the driver or obtain a `DataSource` instance.
3. Invoke the JDBC `DriverManager.getConnection` or `DataSource.getConnection` method to obtain a JDBC `Connection`.
4. Invoke the `ConnectionContext` constructor with the `Connection` as its argument to create the `ConnectionContext` object.

Obtaining a JDBC Connection from an SQLJ ConnectionContext: To do this,

1. Execute an SQLJ connection declaration clause to create a `ConnectionContext` class.
2. Load the driver or obtain a `DataSource` instance.
3. Invoke the `ConnectionContext` constructor with the URL of the driver and any other necessary parameters as its arguments to create the `ConnectionContext` object.
4. Invoke the JDBC `ConnectionContext.getConnection` method to create the JDBC `Connection` object.

See Connect to a data source using SQLJ for more information on SQLJ connections.

Retrieving JDBC result sets using SQLJ iterators: Use the *iterator conversion statement* to manipulate a JDBC result set as an SQLJ iterator. The general form of an iterator conversion statement is:

```
#sql iterator={CAST :result-set};
```

Before you can successfully cast a result set to an iterator, the iterator must conform to the following rules:

- The iterator must be declared as public.
- If the iterator is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must match the data type of the corresponding column in the iterator.
- If the iterator is a named iterator, the name of each accessor method must match the name of a column in the result set. In addition, the data type of the object that an accessor method returns must match the data type of the corresponding column in the result set.

The code in Figure 49 builds and executes a query using a JDBC call, executes an iterator conversion statement to convert the JDBC result set to an SQLJ iterator, and retrieves rows from the result table using the iterator.

```
#sql public iterator ByName(String LastName, Date HireDate);      1
public void HireDates(ConnectionContext connCtx, String whereClause)
{

  ByName nameiter;           // Declare object of ByName class
  Connection conn=connCtx.getConnection();
                             // Create JDBC connection
  Statement stmt = conn.createStatement();                         2
  String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
  query+=whereClause;   // Build the query
  ResultSet rs = stmt.executeQuery(query);                         3
  #sql [connCtx] nameiter = {CAST :rs};                            4
  while (nameiter.next())
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());

  }
  nameiter.close();                                                5
  stmt.close();
}
```

Figure 49. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 49:

1. This SQLJ clause creates the named iterator class ByName, which has accessor methods LastName() and HireDate() that return the data from result table columns LASTNAME and HIREDATE.
2. This statement and the following two statements build and prepare a query for dynamic execution using JDBC.
3. This JDBC statement executes the SELECT statement and assigns the result table to result set rs.
4. This iterator conversion clause converts the JDBC `ResultSet` rs to SQLJ iterator nameiter, and the following statements use nameiter to retrieve values from the result table.
5. The `nameiter.close()` method closes the SQLJ iterator and JDBC `ResultSet` rs.

Generating JDBC ResultSets from SQLJ iterators: Use the getResultSet method to generate a JDBC `ResultSet` from an SQLJ iterator. Every SQLJ iterator has a getResultSet method. After you convert an iterator to a result set, you need to fetch rows using only the result set.

The code in Figure 50 on page 347 generates a positioned iterator for a query, converts the iterator to a result set, and uses JDBC methods to fetch rows from the table.

```
#sql iterator EmpIter(String, java.sql.Date);
{
...
  EmpIter iter=null;
  #sql [connCtx] iter=
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                1
  ResultSet rs=iter.getResultSet();                           2
  while (rs.next())                                           3
  { System.out.println(rs.getString(1) + " was hired in " +
      rs.getDate(2));
  }
  rs.close();                                                 4
}
```

Figure 50. Converting an SQLJ iterator to a JDBC ResultSet

Notes to Figure 50:

1. This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable iter.
2. The getResultSet() method converts iterator iter to `ResultSet` rs.
3. The JDBC getString() and getDate() methods retrieve values from the `ResultSet`. The next() method moves the cursor to the next row in the `ResultSet`.
4. The `rs.close()` method closes the SQLJ iterator as well as the `ResultSet`.

Rules and restrictions for using JDBC ResultSets in SQLJ applications: When you write SQLJ applications that include JDBC result sets, observe the following rules and restrictions:

- You cannot cast a `ResultSet` to an SQLJ iterator if the `ResultSet` and the iterator have different holdability attributes.

  A JDBC `ResultSet` or an SQLJ iterator can remain open after a COMMIT operation. For a JDBC `ResultSet`, this characteristic is controlled by the DB2 Universal JDBC Driver property `resultSetHoldability`. For an SQLJ iterator, this characteristic is controlled by the `with holdability` parameter of the iterator declaration. Casting a `ResultSet` that has holdability to an SQLJ iterator that does not, or casting a `ResultSet` that does not have holdability to an SQLJ iterator that does, is not supported.
- Close a generated `ResultSet` object or the underlying iterator at the end of the program.

  Closing the iterator object from which a `ResultSet` object is generated also closes the `ResultSet` object. Closing the generated `ResultSet` object also closes the iterator object. In general, it is best to close the object that is used last.
- For the DB2 Universal JDBC Driver, which supports scrollable iterators and scrollable and updatable `ResultSets`, the following restrictions apply:
  - Scrollable iterators have the same restrictions as their underlying JDBC `ResultSets`. For example, because scrollable `ResultSets` do not support INSERTs, scrollable iterators do not support INSERTs.
  - You cannot cast a JDBC `ResultSet` that is not updatable to an SQLJ iterator that is updatable.

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

LOBs in SQLJ applications with the DB2 Universal JDBC Driver

With the DB2 Universal JDBC Driver, you can retrieve LOB data into `Clob` or `Blob` host expressions or update CLOB, BLOB, or DBCLOB columns from `Clob` or `Blob` host expressions. You can also declare iterators with `Clob` or `Blob` data types to retrieve data from CLOB, BLOB, or DBCLOB columns.

Retrieving or updating LOB data: To retrieve data from a BLOB column, declare an iterator that includes a data type of `Blob` or `byte[]`. To retrieve data from a CLOB or DBCLOB column, declare an iterator in which the corresponding column has a `Clob` data type.

To update data in a BLOB column, use a host expression with data type `Blob`. To update data in a CLOB or DBCLOB column, use a host expression with data type `Clob`.

LOB locator support: The DB2 Universal JDBC Driver can use LOB locators to retrieve data. To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to `false`. Properties are discussed in Properties for the DB2® Universal JDBC Driver. `fullyMaterializeLobData` has no effect on stored procedure output parameters or LOBs that are fetched using scrollable cursors. You cannot call a stored procedure that has LOB locator parameters. When you fetch from scrollable cursors, JDBC always uses LOB locators to retrieve data from LOB columns.

As in any other language, a LOB locator in a Java application is associated with only one database. You cannot use a single LOB locator to move data between two different databases. To move LOB data between two databases, you need to materialize the LOB data when you retrieve it from a table in the first database and then insert that data into the table in the second database.

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370
- "Java, JDBC, and SQL data types" on page 365

Java data types for retrieving or updating LOB column data in SQLJ applications

When the deferPrepares property is set to true, and the DB2 Universal JDBC Driver processes an uncustomized SQLJ statement that includes host expressions, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

When the JDBC driver cannot immediately determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

When the JDBC driver cannot determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

Input parameters for BLOB columns:

For input parameters for BLOB columns, you can use either of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:

```
java.sql.Blob blobData;
#sql {CALL STORPROC(:IN blobData)};
```

  Before you can use a java.sql.Blob input variable, you need to create a java.sql.Blob object, and then populate that object. For example, if you are using the DB2 Universal JDBC Driver, you can use the DB2-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob` to create a java.sql.Blob object and populate the object with byte[] data:

```
byte[] byteArray = {0, 1, 2, 3};
java.sql.Blob blobData =
  com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob(byteArray);
```

- Use an input parameter of type of `sqlj.runtime.BinaryStream`. A `sqlj.runtime.BinaryStream` object is compatible with a BLOB data type. For this call, you need to specify the exact length of the input data:

```
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream(byteData);
int numBytes = byteData.length;
sqlj.runtime.BinaryStream binStream =
  new sqlj.runtime.BinaryStream(byteStream, numBytes);
#sql {CALL STORPROC(:IN binStream)};
```

  You cannot use this technique for input/output parameters.

Output parameters for BLOB columns:

For output or input/output parameters for BLOB columns, you can use the following technique:

- Declare the output parameter or input/output variable with a java.sql.Blob data type:

```
java.sql.Blob blobData = null;
#sql CALL STORPROC (:OUT blobData)};
```

```
java.sql.Blob blobData = null;
#sql CALL STORPROC (:INOUT blobData)};
```

Input parameters for CLOB columns:

For input parameters for CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:

```
#sql CALL STORPROC(:IN clobData)};
```

  Before you can use a java.sql.Clob input variable, you need to create a java.sql.Clob object, and then populate that object. For example, if you are using the DB2 Universal JDBC Driver, you can use the DB2-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob` to create a java.sql.Clob object and populate the object with String data:

```
String stringVal = "Some Data";
java.sql.Clob clobData =
  com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob(stringVal);
```

- Use one of the following types of stream input parameters:
  - A `sqlj.runtime.CharacterStream` input parameter:

```
java.lang.String charData;
java.io.StringReader reader = new java.io.StringReader(charData);
sqlj.runtime.CharacterStream charStream =
  new sqlj.runtime.CharacterStream (reader, charData.length);
#sql {CALL STORPROC(:IN charStream)};
```

– A `sqlj.runtime.UnicodeStream` parameter, for Unicode UTF-16 data:

```
byte[] charDataBytes = charData.getBytes("UnicodeBigUnmarked");
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream(charDataBytes);
sqlj.runtime.UnicodeStream uniStream =
  new sqlj.runtime.UnicodeStream(byteStream, charDataBytes.length );
#sql {CALL STORPROC(:IN uniStream)};
```

– A `sqlj.runtime.AsciiStream` parameter, for ASCII data:

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
  new java.io.ByteArrayInputStream (charDataBytes);
sqlj.runtime.AsciiStream asciiStream =
  new sqlj.runtime.AsciiStream (byteStream, charDataBytes.length);
#sql {CALL STORPROC(:IN asciiStream)};
```

For these calls, you need to specify the exact length of the input data. You cannot use this technique for input/output parameters.

- Use a java.lang.String input parameter:

```
java.lang.String charData;
#sql {CALL STORPROC(:IN charData)};
```

Output parameters for CLOB columns:

For output our input/output parameters for CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob output variable, which is an exact match for a CLOB column:

```
java.sql.Clob clobData = null;
#sql CALL STORPROC(:OUT clobData)};
```

- Use a java.lang.String output variable:

```
java.lang.String charData = null;
#sql CALL STORPROC(:OUT charData)};
```

This technique should be used only if you know that the length of the retrieved data is less than or equal to 32KB. Otherwise, the data is truncated.

Output parameters for DBCLOB columns:

DBCLOB output or input/output parameters for stored procedures are not supported.

Related concepts:

- "LOBs in SQLJ applications with the DB2 Universal JDBC Driver" on page 348

Related reference:

- "Java, JDBC, and SQL data types" on page 365

ROWIDs in SQLJ with the DB2 Universal JDBC Driver

DB2® UDB for z/OS® and DB2 UDB for iSeries™ support the ROWID data type for a column in a DB2 table. A ROWID is a value that uniquely identifies a row in a table.

If you use ROWIDs in SQLJ programs, you need to customize those programs.

The DB2 Universal JDBC Driver provides the DB2-only class `com.ibm.db2.jcc.DB2RowID` that you can use in iterators and in CALL statement parameters. For an iterator, you can also use the byte[] object type to retrieve ROWID values.

Figure 51 shows an example of an iterator that is used to select values from a ROWID column:

```
#sql iterator PosIter(int,String,com.ibm.db2.jcc.DB2RowId);
                                // Declare positioned iterator
                                // for retrieving ITEM_ID (INTEGER),
                                // ITEM_FORMAT (VARCHAR), and ITEM_ROWID (ROWID)
                                // values from table ROWIDTAB
{
  PosIter positrowid;           // Declare object of PosIter class
  com.ibm.db2.jcc.DB2RowId rowid = null;
  int id = 0;
  String i_fmt = null;
                                // Declare host expressions
  #sql [ctxt] positrowid =
    {SELECT ITEM_ID, ITEM_FORMAT, ITEM_ROWID FROM ROWIDTAB
       WHERE ITEM_ID=3};
                                // Assign the result table of the SELECT
                                // to iterator object positrowid
  #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the first row
  while (!positrowid.endFetch())
                                // Check whether the FETCH returned a row
    {System.out.println("Item ID " + id + " Item format " +
       i_fmt + " Item ROWID ");
     printBytes(rowid.getBytes());
                                // Use the DB2-only method getBytes to
                                // convert the value to bytes for printing
      #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the next row
    }
  positrowid.close();           // Close the iterator
}
```

Figure 51. Example of using an iterator to retrieve ROWID values

Figure 52 on page 352 shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
com.ibm.db2.jcc.DB2RowId in_rowid = rowid;
com.ibm.db2.jcc.DB2RowId out_rowid = null;
com.ibm.db2.jcc.DB2RowId inout_rowid = rowid;
                                    // Declare an input, output, and
                                    // input/output ROWID parameter
...
#sql [myConnCtx] {CALL SP_ROWID(:IN in_rowid,
                              :OUT out_rowid,
                              :INOUT inout_rowid)};
                                    // Call the stored procedure
System.out.println("Parameter values from SP_ROWID call: ");
System.out.println("Output parameter value ");
printBytes(out_rowid.getBytes());
                                  // Use the DB2-only method getBytes to
                                  // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_rowid.getBytes());
```

Figure 52. Example of calling a stored procedure with a ROWID parameter

Related reference:

- "Java, JDBC, and SQL data types" on page 365

Distinct types in SQLJ applications

In DB2®, a distinct type is a user-defined data type that is internally represented as a built-in SQL data type. You create a distinct type by executing the SQL statement CREATE DISTINCT TYPE.

In an SQLJ program, you can create a distinct type using the CREATE DISTINCT TYPE statement in an executable clause. You can also use CREATE TABLE in an executable clause to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java™ identifiers with data types that correspond to the built-in types on which the distinct types are based.

The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```
String empNumVar;
int shoeSizeVar;
...
#sql [myConnCtx] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};
                                          // Create distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {CREATE TABLE EMP_SHOE
  (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)};
                                          // Create table using distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {INSERT INTO EMP_SHOE
  VALUES('000010',6)};                    // Insert a row in the table
#sql [myConnCtx] {COMMIT}; // Commit the INSERT
#sql [myConnCtx] {SELECT EMPNO, EMP_SHOE_SIZE
  INTO :empNumVar, :shoeSizeVar
  FROM EMP_SHOE};                         // Retrieve the row
System.out.println("Employee number: " + empNumVar +
  " Shoe size: " + shoeSizeVar);
```

Figure 53. Defining and using a distinct type

Related reference:

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

Controlling the execution of SQL statements in SQLJ

You can use selected methods of the SQLJ `ExecutionContext` class to control or monitor the execution of SQL statements. Selected sqlj.runtime classes and interfaces describes those methods.

To use `ExecutionContext` methods, follow these steps:

1. Acquire an *execution context*.

   There are two ways to acquire an execution context:

   - Acquire the default execution context from the connection context. For example:

     ```
     ExecutionContext execCtx = connCtx.getExecutionContext();
     ```

   - Create a new execution context by invoking the contructor for `ExecutionContext`. For example:

     ```
     ExecutionContext execCtx=new ExecutionContext();
     ```

2. Associate the execution context with an SQL statement.

   To do that, specify an execution context after the connection context in the execution clause that contains the SQL statement. For example:

   ```
   #sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
   ```

3. Invoke `ExecutionContext` methods.

   Some ExecutionContext methods are applicable before the associated SQL statement is executed, and some are applicable only after their associated SQL statement is executed.

   For example, you can use method `getUpdateCount` to count the number of rows that are deleted by a DELETE statement after you execute the DELETE statement:

   ```
   #sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
   System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
   ```

Related reference:

- "Selected sqlj.runtime classes and interfaces" on page 407

Retrieving multiple result sets from a stored procedure in an SQLJ application

Some stored procedures return one or more result sets to the calling program. To retrieve the rows from those result sets, you execute these steps:

1. Acquire an execution context for retrieving the result set from the stored procedure.
2. Associate the execution context with the CALL statement for the stored procedure.

   Do not use this execution context for any other purpose until you have retrieved and processed the last result set.
3. For each result set:
   a. Use the `ExecutionContext` method `getNextResultSet` to retrieve the result set.
   b. If you do not know the contents of the result set, use `ResultSetMetaData` methods to retrieve this information.
   c. Use an SQLJ result set iterator or JDBC `ResultSet` to retrieve the rows from the result set.

Result sets are returned to the calling program in the same order that their cursors are opened in the stored procedure. When there are no more result sets to retrieve, getNextResultSet returns a null value.

`getNextResultSet` has two forms:

```
getNextResultSet();
getNextResultSet(int current);
```

When you invoke the first form of `getNextResultSet`, SQLJ closes the currently-open result set and advances to the next result set. When you invoke the second form of `getNextResultSet`, the value of *current* indicates what SQLJ does with the currently-open result set before it advances to the next result set:

java.sql.Statement.CLOSE_CURRENT_RESULT
Specifies that the current `ResultSet` object is closed when the next `ResultSet` object is returned.

java.sql.Statement.KEEP_CURRENT_RESULT
Specifies that the current `ResultSet` object stays open when the next `ResultSet` object is returned.

java.sql.Statement.CLOSE_ALL_RESULTS
Specifies that all open `ResultSet` objects are closed when the next `ResultSet` object is returned.

The second form of `getNextResultSet` requires JDK 1.4 or later.

The following code calls a stored procedure that returns multiple result sets. For this example, it is assumed that the caller does not know the number of result sets to be returned or the contents of those result sets. It is also assumed that autoCommit is false. The numbers to the right of selected statements correspond to the previously-described steps.

```
ExecutionContext execCtx=myConnCtx.getExecutionContext();      1
#sql [myConnCtx, execCtx] {CALL MULTRSSP()};                   2
           // MULTRSSP returns multiple result sets
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null)              3a
{
  ResultSetMetaData rsmeta=rs.getMetaData();                   3b
  int numcols=rsmeta.getColumnCount();
  while (rs.next())                                            3c
  {
    for (int i=1; i<=numcols; i++)
    {
      String colval=rs.getString(i);
      System.out.println("Column " + i + "value is " + colval);
    }
  }
}
```

Figure 54. Retrieving result sets from a stored procedure

Making batch updates in SQLJ applications

The DB2 Universal JDBC Driver supports batch updates in SQLJ. With batch updates, instead of updating rows of a DB2® table one at a time, you can direct SQLJ to execute a group of updates at the same time. You can include the following types of statements in a batch update:

- Searched INSERT, UPDATE, or DELETE statements
- CREATE, ALTER, DROP, GRANT, or REVOKE statements
- CALL statements with input parameters only

Unlike JDBC, SQLJ allows heterogeneous batches that contain statements with input parameters or host expressions. You can therefore combine instances of the same statement, different statements, statements with input parameters or host expressions, and statements with no input parameters or host expressions in the same SQLJ statement batch.

The basic steps for creating, executing, and deleting a batch of statements are:

1. Disable `AutoCommit` for the connection.
2. Acquire an execution context.

   All statements that execute in a batch must use this execution context.
3. Invoke the `ExecutionContext.setBatching(true)` method to create a batch.

   Subsequent batchable statements that are associated with the execution context that you created in step 2 are added to the batch for later execution.

   If you want to batch sets of statements that are not batch compatible in parallel, you need to create an execution context for each set of batch compatible statements.
4. Include SQLJ executable clauses for SQL statements that you want to batch.

   These clauses must include the execution context that you created in step 2.

   If an SQLJ executable clause has input parameters or host expressions, you can include the statement in the batch multiple times with different values for the input parameters or host expressions.

To determine whether a statement was added to an existing batch, was the first statement in a new batch, or was executed inside or outside a batch, invoke the `ExecutionContext.getUpdateCount` method. This method returns one of the following values:

`ExecutionContext.ADD_BATCH_COUNT`
This is a constant that is returned if the statement was added to an existing batch.

`ExecutionContext.NEW_BATCH_COUNT`
This is a constant that is returned if the statement was the first statement in a new batch.

`ExecutionContext.EXEC_BATCH_COUNT`
This is a constant that is returned if the statement was part of a batch, and the batch was executed.

Other integer
This value is the number of rows that were updated by the statement. This value is returned if the statement was executed rather than added to a batch.

5. Execute the batch explicitly or implicitly.
   - Invoke the `ExecutionContext.executeBatch` method to execute the batch explicitly.

     `executeBatch` returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch.
   - Alternatively, a batch executes implicitly under the following circumstances:
     – You include a batchable statement in your program that is not compatible with statements that are already in the batch. In this case, SQLJ executes the statements that are already in the batch and creates a new batch that includes the incompatible statement. SQLJ also executes the statement that is not compatible with the statements in the batch.
     – You include a statement in your program that is not batchable. In this case, SQLJ executes the statements that are already in the batch. SQLJ also executes the statement that is not batchable.
     – After you invoke the `ExecutionContext.setBatchLimit(n)` method, you add a statement to the batch that brings the number of statements in the batch to *n* or greater. *n* can have one of the following values:

       `ExecutionContext.UNLIMITED_BATCH`
       This constant indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking `setBatchLimit`.

       `ExecutionContext.AUTO_BATCH`
       This constant indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

       Positive integer
       When this number of statements have been added to the batch, SQLJ executes the batch implicitly. However, the batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

To determine the number of rows that were updated by a batch that was executed implicitly, invoke the `ExecutionContext.getBatchUpdateCounts` method. `getBatchUpdateCounts` returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch. Each array element can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

Other integer
This value is the number of rows that were updated by the statement.

6. Optionally, when all statements have been added to the batch, disable batching.

   Do this by invoking the `ExecutionContext.setBatching(false)` method. When you disable batching, you can still execute the batch implicitly or explicitly, but no more statements are added to the batch. Disabling batching is useful when a batch already exists, and you want to execute a batch compatible statement, rather than adding it to the batch.

   If you want to clear a batch without executing it, invoke the `ExecutionContext.cancel` method.

7. If batch execution was implicit, perform a final, explicit `executeBatch` to ensure that all statements have been executed.

Example of a batch update: In the following code fragment, raises are given to all managers by performing UPDATEs in a batch. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql iterator GetMgr(String);       // Declare positioned iterator
{
  GetMgr deptiter;                  // Declare object of GetMgr class
  String mgrnum = null;             // Declare host variable for manager number
  int raise = 400®;                  // Declare raise amount
  int currentSalary;                // Declare current salary
  String url, username, password;   // Declare url, user ID, password
  ...
  TestContext c1 = new TestContext (url, username, password, false);  1
  ExecutionContext ec = new ExecutionContext();                       2
  ec.setBatching(true);                                               3

  #sql [c1] deptiter =
    {SELECT MGRNO FROM DEPARTMENT};
                                    // Assign the result table of the SELECT
                                    // to iterator object deptiter
  #sql {FETCH :deptiter INTO :mgrnum};
                                    // Retrieve the first manager number
  while (!deptiter.endFetch()) {    // Check whether the FETCH returned a row
    #sql [c1]
      {SELECT SALARY INTO :currentSalary FROM EMPLOYEE
        WHERE EMPNO=:mgrnum};
    #sql [c1, ec]                                                     4
      {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
        WHERE EMPNO=:mgrnum};
    #sql {FETCH :deptiter INTO :mgrnum };
                                    // Fetch the next row
  }
  ec.executeBatch();                                                  5
  ec.setBatching(false);                                              6
  #sql [c1] {COMMIT};
  deptiter.close();                 // Close the iterator
  ec.close();                       // Close the execution context
  c1.close();                       // Close the connection
}
```

Figure 55. Performing a batch update

When an error occurs during execution of a statement in a batch, the remaining statements are executed, and a `BatchUpdateException` is thrown after all the statements in the batch have executed. See Make batch updates in a JDBC application for information on how to process a `BatchUpdateException`.

To obtain information about warnings, use the `Statement.getWarnings` method on the object on which you ran the `executeBatch` method. You can then retrieve an error description, SQLSTATE, and error code for each `SQLWarning` object.

When a batch is executed implicitly because the program contains a statement that cannot be added to the batch, the batch is executed before the new statement is processed. If an error occurs during execution of the batch, the statement that caused the batch to execute does not execute.

Recommendation: Turn autocommit off when you do batch updates so that you can control whether to commit changes to already-executed statements when an error occurs during batch execution.

Related tasks:

- "Making batch updates in JDBC applications" on page 304
- "Connecting to a data source using SQLJ" on page 322
- "Controlling the execution of SQL statements in SQLJ" on page 353

Related reference:

- "Selected sqlj.runtime classes and interfaces" on page 407

Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application

SQLJ allows iterators to be passed between methods as variables. An iterator that is used for a positioned UPDATE or DELETE can be identified only at runtime. The same SQLJ positioned UPDATE or DELETE statement can be used with different iterators at runtime. If you specify a value of YES for -staticpositioned when you customize your SQLJ application as part of the program preparation process, the SQLJ customizer prepares positioned UPDATE or DELETE statements to execute statically. In this case, the customizer must determine which iterators belong with which positioned UPDATE or DELETE statements. The SQLJ customizer does this by matching iterator data types to data types in the UPDATE or DELETE statements. However, if there is not a unique mapping of tables in UPDATE or DELETE statements to iterator classes, the SQLJ customizer cannot determine exactly which iterators and UPDATE or DELETE statements go together. The SQLJ customizer must arbitrarily pair iterators with UPDATE or DELETE statements, which can sometimes result in SQL errors. The following code fragments illustrate this point.

```
#sql iterator GeneralIter ( String );

  public static void main ( String args[] )
  {
...
    GeneralIter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };

    GeneralIter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };
...

    doUpdate ( iter1 );
  }

  public static void doUpdate ( GeneralIter iter )
  {
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
  }
```

Figure 56. Static positioned UPDATE that succeeds

In this example, only one iterator is defined. Two instances of that iterator are defined, and each is associated with a different SELECT statement that retrieves data from a different table. Because the iterator is passed to method `doUpdate` as a variable, it is impossible to know until run time which of the iterator instances is used for the positioned UPDATE. The DB2® bind process uses the first iterator instance, `iter1`, when it binds the DB2 plan. At run time, if `iter1` is passed to the `doUpdate` method, as shown in Figure 56, the UPDATE succeeds because `iter1` and the UPDATE statement both use `TABLE1`.

If the program is written in a slightly different way, as shown in Figure 57 on page 360, the DB2 bind fails, even though the program appears to be valid.

```
#sql iterator GeneralIter ( String );

  public static void main ( String args[] )
  {
...
    GeneralIter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };
    GeneralIter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };

...

    doUpdate ( iter1 );
  }

  public static void doUpdate ( GeneralIter iter )
  {
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
  }
```

Figure 57. Static positioned UPDATE that fails at bind time

In this case, the DB2 bind process associates `iter2` with the positioned UPDATE because `iter2` comes first in the program. When DB2 binds the plan for the program, the bind fails with SQLCODE -509 because `iter2` uses `TABLE2` and the UPDATE uses `TABLE1`. However, if this program is allowed to bind successfully, and you pass `iter1` to the `doUpdate` method, the program runs successfully.

You can avoid a bind time error for a program like the one in Figure 57 by specifying the DB2 BIND option SQLERROR(CONTINUE). However, this technique has the drawback that it causes DB2 to build a package, regardless of the SQL errors that are in the program. A better technique is to write the program so that there is a one-to-one mapping between tables in positioned UPDATE or DELETE statements and iterator classes. Figure 58 on page 361 shows an example of how to do this.

```
#sql iterator Table2Iter(String);
#sql iterator Table1Iter(String);
  public static void main ( String args[] )
  {
...
    Table2Iter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };

    Table1Iter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };
...

    doUpdate(iter1);

  }

 public static void doUpdate ( Table1Iter iter )
  {
    ...
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
    ...
  }
 public static void doUpdate ( Table2Iter iter )
  {
    ...
    #sql [ctxt] { UPDATE TABLE2 ... WHERE CURRENT OF :iter };
    ...
  }
```

Figure 58. Static positioned UPDATE that succeeds regardless of iterator order

With this method of coding, each iterator class is associated with only one table. Therefore, the DB2 bind process can always associate the positioned UPDATE statement with a valid iterator.

Related tasks:

- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336

Related reference:

- "db2sqljcustomize - DB2 SQLJ Profile Customizer Command" in the *Command Reference*

Using scrollable iterators in an SQLJ application

In addition to moving forward, one row at a time, through a result table, you might want to move backward or go directly to a specific row. The DB2 Universal JDBC Driver provides this capability.

An iterator in which you can move forward, backward, or to a specific row is called a *scrollable iterator*. A scrollable iterator in SQLJ is equivalent to the result table of a DB2® cursor that is declared as SCROLL.

Like a scrollable cursor, a scrollable iterator can be *insensitive* or *sensitive*. A sensitive scrollable iterator can be *static* or *dynamic*. Insensitive means that changes to the underlying table after the iterator is opened are not visible to the iterator. Insensitive iterators are read-only. Sensitive means that changes that the iterator or other processes make to the underlying table are visible to the iterator.

If a scrollable iterator is static, the size of the result table and the order of the rows in the result table do not change after the iterator is opened. This means that you cannot insert into result tables, and if you delete a row of a result table, a delete hole occurs. If you update a row of the result table so that the row no longer qualifies for the result table, an update hole occurs. Fetching from a hole results in an `SQLException`.

If a scrollable iterator is dynamic, the size of the result table and the order of the rows in the result table can change after the iterator is opened. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by the same application process are immediately visible. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by other application processes are visible after the changes are committed.

To create and use a scrollable iterator, you need to follow these steps:

1. Specify an iterator declaration clause that includes the following clauses:
   - `implements sqlj.runtime.Scrollable`

     This indicates that the iterator is scrollable.
   - `with (sensitivity=INSENSITIVE|SENSITIVE)` or `with (sensitivity=SENSITIVE, dynamic=true|false)`

     `sensitivity=INSENSITIVE|SENSITIVE` indicates whether update or delete operations on the underlying table can be visible to the iterator. The default sensitivity is `INSENSITIVE`.

     `dynamic=true|false` indicates whether the size of the result table or the order of the rows in the result table can change after the iterator is opened. The default value of dynamic is `false`.

   The iterator can be a named or positioned iterator. For example, the following iterator declaration clause declares a positioned, sensitive, dynamic, scrollable iterator:

   ```
   #sql public iterator ByPos
     implements sqlj.runtime.Scrollable
     with (sensitivity=SENSITIVE, dynamic=true)  (String);
   ```

   The following iterator declaration clause declares a named, insensitive, scrollable iterator:

   ```
   #sql public iterator ByName
     implements sqlj.runtime.Scrollable
     with (sensitivity=INSENSITIVE) (String EmpNo);
   ```

   Restriction: You cannot use a scrollable iterator to select columns with the following data types from a table on a DB2 UDB for Linux, UNIX, and Windows server:
   - LONG VARCHAR
   - LONG VARGRAPHIC
   - DATALINK
   - BLOB
   - CLOB
   - A distinct type that is based on any of the previous data types in this list
   - A structured type
2. Create an iterator object, which is an instance of your iterator class.
3. If you want to give the SQLJ runtime environment a hint about the initial fetch direction, use the `setFetchDirection(int direction)` method. *direction* can be `FETCH_FORWARD` or `FETCH_REVERSE`. If you do not invoke `setFetchDirection`, the fetch direction is `FETCH_FORWARD`.

4. For each row that you want to access:
   - For a named iterator, perform the following steps:
     a. Position the cursor using one of the methods listed in Table 38.

Table 38. sqlj.runtime.Scrollable methods for positioning a scrollable cursor

Method	Positions the cursor
`first()`	On the first row of the result table
`last()`	On the last row of the result table
`previous()`[1]	On the previous row of the result table
`next()`	On the next row of the result table
`absolute(int n)`[2]	If *n*>0, on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, on row *m+n+1* of the result table.
`relative(int n)`[3]	If *n*>0, on the row that is *n* rows after the current row. If *n*<0, on the row that is *n* rows before the current row. If *n*=0, on the current row.
`afterLast()`	After the last row in the result table
`beforeFirst()`	Before the first row in the result table

Notes:

1. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
2. If the absolute value of *n* is greater than the number of rows in the result table, this method positions the cursor after the last row if *n* is positive, or before the first row if *n* is negative.
3. Suppose that *m* is the number of rows in the result table and *x* is the current row number in the result table. If *n*>0 and *x+n*>*m*, the iterator is positioned after the last row. If *n*<0 and *x+n*<1, the iterator is positioned before the first row.

     b. If you need to know the current cursor position, use the `getRow`, `isFirst`, `isLast`, `isBeforeFirst`, or `isAfterLast` method to obtain this information. If you need to know the current fetch direction, invoke the `getFetchDirection` method.
     c. Use accessor methods to retrieve the current row of the result table.
     d. If update or delete operations by the iterator or by other means are visible in the result table, invoke the `getWarnings` method to check whether the current row is a hole.
   - For a positioned iterator, perform the following steps:
     a. Use a FETCH statement with a fetch orientation clause to position the iterator and retrieve the current row of the result table. Table 39 lists the clauses that you can use to position the cursor.

Table 39. FETCH clauses for positioning a scrollable cursor

Method	Positions the cursor
FIRST	On the first row of the result table
LAST	On the last row of the result table
PRIOR[1]	On the previous row of the result table
NEXT	On the next row of the result table
ABSOLUTE(*n*)[2]	If *n*>0, on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, on row *m+n+1* of the result table.

Table 39. FETCH clauses for positioning a scrollable cursor (continued)

Method	Positions the cursor
RELATIVE(n)[3]	If n>0, on the row that is n rows after the current row. If n<0, on the row that is n rows before the current row. If n=0, on the current row.
AFTER[4]	After the last row in the result table
BEFORE[4]	Before the first row in the result table

Notes:

1. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
2. If the absolute value of n is greater than the number of rows in the result table, this method positions the cursor after the last row if n is positive, or before the first row if n is negative.
3. Suppose that m is the number of rows in the result table and x is the current row number in the result table. If n>0 and $x+n>m$, the iterator is positioned after the last row. If n<0 and $x+n$<1, the iterator is positioned before the first row.
4. Values are not assigned to host expressions.

   b. If update or delete operations by the iterator or by other means are visible in the result table, invoke the `getWarnings` method to check whether the current row is a hole.

5. Invoke the `close` method to close the iterator.

For example, the following code demonstrates how to use a named iterator to retrieve the employee number and last name from all rows from the employee table in reverse order. The numbers to the right of selected statements correspond to the previously-described steps.

```
#sql  iterator ScrollIter implements sqlj.runtime.Scrollable          1
  (String EmpNo, String LastName);
{
  ScrollIter scrliter;                                                2
  #sql [ctxt]
  scrliter={SELECT EMPNO, LASTNAME FROM EMPLOYEE};
  scrliter.afterLast();
  while (scrliter.previous())                                         4a
  {
    System.out.println(scrliter.EmpNo() + " "                         4c
      + scrliter.LastName());
  }
  scrliter.close();                                                   5
}
```

Figure 59. Using scrollable iterators

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 331

Related tasks:

- "Using a named iterator in an SQLJ application" on page 332
- "Using a positioned iterator in an SQLJ application" on page 334

Chapter 17. JDBC and SQLJ reference

The sections that follow contain reference information about JDBC methods and SQLJ clauses.

Java, JDBC, and SQL data types

The following tables summarize the mappings of Java data types to JDBC and SQL data types for a DB2 UDB for Linux, UNIX and Windows system.

Table 40 summarizes the mappings of Java data types to DB2 data types for `PreparedStatement.setXXX` or `ResultSet.updateXXX` methods in JDBC programs, and for input host expressions in SQLJ programs. When more than one Java data type is listed, the first data type is the recommended data type.

Table 40. Mappings of Java data types to DB2 data types for updating DB2 tables

Java data type	SQL data type
short, boolean[1], byte[1]	SMALLINT
int, java.lang.Integer	INTEGER
long, java.lang.Long	DECIMAL(19,0)[2]
long, java.lang.Long	BIGINT[3]
float, java.lang.Float	REAL
double, java.lang.Double	DOUBLE
java.math.BigDecimal	DECIMAL(*p*,*s*)[4]
java.lang.String	CHAR(*n*)[5]
java.lang.String	GRAPHIC(*m*)[6]
java.lang.String	VARCHAR(*n*)[7]
java.lang.String	VARGRAPHIC(*m*)[8]
java.lang.String	CLOB(*n*)[9]
byte[]	CHAR(*n*) FOR BIT DATA[5]
byte[]	VARCHAR(*n*) FOR BIT DATA[7]
byte[]	BLOB(*n*)[9,10]
byte[]	ROWID
java.sql.Blob	BLOB(*n*)[10]
java.sql.Clob	CLOB(*n*)[10]
java.sql.Clob	DBCLOB(*m*)[11]
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.io.ByteArrayInputStream	BLOB(*n*)[10]
java.io.StringReader	CLOB(*n*)[10]
java.io.ByteArrayInputStream	CLOB(*n*)[10]
com.ibm.db2.jcc.DB2RowID	ROWID

Table 40. Mappings of Java data types to DB2 data types for updating DB2 tables (continued)

Java data type	SQL data type
java.net.URL	DATALINK[12]

Notes:

1. DB2 has no exact equivalent for the Java boolean or byte data types, but the best fit is SMALLINT.
2. DB2 UDB in the OS/390 or z/OS environment has no exact equivalent for the Java long or java.lang.Long data types, but the best fit is DECIMAL(19,0).
3. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows.
4. *p* is the decimal precision and *s* is the scale of the DB2 column.

   You should design financial applications so that java.math.BigDecimal columns map to DECIMAL columns. If you know the precision and scale of a DECIMAL column, updating data in the DECIMAL column with data in a java.math.BigDecimal variable results in better precision and performance than using other combinations of data types.
5. *n*<=254.
6. *m*<=127.
7. *n*<=32672.
8. *m*<=16336.
9. This mapping is valid only if DB2 can determine the data type of the column.
10. *n*<=2147483647.
11. *m*<=1073741823.
12. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 41 summarizes the mappings of DB2 data types to Java data types for `ResultSet.getXXX` methods in JDBC programs, and for iterators in SQLJ programs. This table does not list Java numeric wrapper object types, which are retrieved using `ResultSet.getObject`.

Table 41. Mappings of DB2 data types to Java data types for retrieving data from DB2 tables

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
SMALLINT	short	byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGINT[1]	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
REAL	float	long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String
DOUBLE	double	long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String
CHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader

Table 41. Mappings of DB2 data types to Java data types for retrieving data from DB2 tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
VARCHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CHAR(*n*) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
VARCHAR(*n*) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
GRAPHIC(*m*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARGRAPHIC(*m*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CLOB(*n*)	java.sql.Clob	java.lang.String
BLOB(*n*)	java.sql.Blob	byte[][3]
DBCLOB(*m*)	No exact equivalent. Use java.sql.Clob.	
ROWID	com.ibm.db2.jcc.DB2RowID	byte[]
DATE	java.sql.Date	java.sql.String, java.sql.Timestamp
TIME	java.sql.Time	java.sql.String, java.sql.Timestamp
TIMESTAMP	java.sql.Timestamp	java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp
DATALINK	java.net.URL[4]	

Notes:

1. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows.
2. You should design financial applications so that DECIMAL columns map to java.math.BigDecimal columns. If you know the precision and scale of a DECIMAL column, retrieving data from that column into a java.math.BigDecimal variable results in better precision and performance than using other combinations of data types.
3. This mapping is valid only if DB2 can determine the data type of the column.
4. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 42 on page 368 summarizes mappings of Java data types to JDBC data types and DB2 data types for user-defined function and stored procedure parameters. The mappings of Java data types to JDBC data types are for `CallableStatement.registerOutParameter` methods in JDBC programs. The mappings of Java data types to DB2 data types are for parameters in stored procedure or user-defined function invocations.

If more than one Java data type is listed in Table 42 on page 368, the first data type is the **recommended** data type.

Table 42. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

Java data type	JDBC data type	SQL data type
boolean[1]	BIT	SMALLINT
byte[1]	TINYINT	SMALLINT
short, java.lang.Integer	SMALLINT	SMALLINT
int, java.lang.Integer	INTEGER	INTEGER
long	BIGINT	BIGINT[2]
float, java.lang.Float	REAL	REAL
float, java.lang.Float	FLOAT	REAL
double, java.lang.Double	DOUBLE	DOUBLE
java.math.BigDecimal	NUMERIC	DECIMAL
java.math.BigDecimal	DECIMAL	DECIMAL
java.lang.String	CHAR	CHAR
java.lang.String	CHAR	GRAPHIC
java.lang.String	VARCHAR	VARCHAR
java.lang.String	VARCHAR	VARGRAPHIC
java.lang.String	LONGVARCHAR	VARCHAR
java.lang.String	VARCHAR	CLOB(*n*)
java.lang.String	LONGVARCHAR	CLOB(*n*)
java.lang.String	CLOB	CLOB(*n*)
byte[]	BINARY	CHAR FOR BIT DATA
byte[]	VARBINARY	VARCHAR FOR BIT DATA
byte[]	LONGVARBINARY	VARCHAR FOR BIT DATA
byte[]	VARBINARY	BLOB(*n*)[3]
byte[]	LONGVARBINARY	BLOB(*n*)[3]
java.sql.Date	DATE	DATE
java.sql.Time	TIME	TIME
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Clob	CLOB	DBCLOB
java.io.ByteArrayInputStream	None	BLOB(*n*)
java.io.StringReader	None	CLOB(*n*)
java.io.ByteArrayInputStream	None	CLOB(*n*)
com.ibm.db2.jcc.DB2RowID	com.ibm.db2.jcc.DB2Types.ROWID	ROWID
java.net.URL	DATALINK	DATALINK[4]

Table 42. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

Java data type	JDBC data type	SQL data type

Notes:

1. A stored procedure or user-defined function that is defined with a SMALLINT parameter can be invoked with a boolean or byte parameter. However, this is not recommended.
2. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows servers. For Java applications that connect from a DB2 UDB Version 8.1 client to a DB2 UDB Version 7 server, when the `CallableStatement.getObject` method is used to retrieve a BIGINT value, a `java.math.BigDecimal` object is returned.
3. This mapping is valid only if DB2 can determine the data type of the column.
4. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 43 summarizes mappings of the SQL parameter data types in a CREATE PROCEDURE or CREATE FUNCTION statement to the data types in the corresponding Java stored procedure or user-defined function method.

For DB2 UDB for Linux, UNIX and Windows, if more than one Java data type is listed for an SQL data type, only the **first** Java data type is valid.

For DB2 UDB in the OS/390 or z/OS environment, if more than one Java data type is listed, and you use a data type other than the first data type as a method parameter, you need to include a method signature in the EXTERNAL clause of your CREATE PROCEDURE or CREATE FUNCTION statement that specifies the Java data types of the method parameters.

Table 43. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program

SQL data type in CREATE PROCEDURE or CREATE FUNCTION	Data type in Java stored procedure or user-defined function method
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
BIGINT[1]	long
REAL	float, java.lang.Float
DOUBLE	double, java.lang.Double
DECIMAL	java.math.BigDecimal
CHAR	java.lang.String
GRAPHIC	java.lang.String
VARCHAR	java.lang.String
VARGRAPHIC	java.lang.String
CHAR FOR BIT DATA	byte[]
VARCHAR FOR BIT DATA	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
DBCLOB	java.sql.Clob

Table 43. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program (continued)

SQL data type in CREATE PROCEDURE or CREATE FUNCTION	Data type in Java stored procedure or user-defined function method
ROWID	com.ibm.db2.jcc.DB2Types.ROWID
DATALINK	java.net.URL[2]

Notes:

1. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows servers.
2. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426

Properties for the DB2 Universal JDBC Driver

Properties define how the connection to a particular data source should be made. Unless otherwise noted, properties can be set for a `DataSource` object or for a `Connection` object. Properties can be set in one of the following ways:

- Using `setXXX` methods

  Properties are applicable to the following DB2-specific implementations that inherit from `com.ibm.db2.jcc.DB2BaseDataSource`:
  - `com.ibm.db2.jcc.DB2SimpleDataSource`
  - `com.ibm.db2.jcc.DB2DataSource`
  - `com.ibm.db2.jcc.DB2ConnectionPoolDataSource`
  - `com.ibm.db2.jcc.DB2XADataSource`

  See Summary of DB2 Universal JDBC Driver extensions to JDBC for a summary of the property names and data types.
- In a `java.util.Properties` value in the *info* parameter of a `DriverManager.getConnection` call, as shown in Connect to a data source using the DriverManager interface with the DB2 Universal JDBC Driver.
- In a `java.lang.String` value in the *url* parameter of a DriverManager.getConnection call, as shown in Connect to a data source using the DriverManager interface with the DB2 Universal JDBC Driver.

The properties are:

activeServerListJNDIName
Identifies a JNDI reference to a `DB2ActiveServerList` instance in a JNDI repository of alternate server information. If the value of activeServerListJNDIName is not null, connections can failover to an alternate server that is specified in the `DB2ActiveServerList` instance that is referenced by the value. If activeServerListJNDIName is null, connections do not failover using alternate server information from a JNDI repository.

clientAccountingInformation
Specifies accounting information for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

clientApplicationInformation
Specifies application information for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

clientUser
Specifies the current client user name for the connection. This information is for client accounting purposes. Unlike the JDBC connection user name, this value can change during a connection. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes.

clientWorkstation
Specifies the workstation name for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this value, but a Java `null` value is not valid.

cliSchema
Specifies the schema of the DB2 shadow catalog tables or views that are searched when an application invokes a `DatabaseMetaData` method.

currentFunctionPath
Specifies the SQL path that is used to resolve unqualified data type names and function names in SQL statements that are in JDBC programs. The data type of this property is String. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 254 bytes. The value is a comma-separated list of schema names. Those names can be ordinary or delimited identifiers.

currentLockTimeout
Directs DB2 UDB for Linux, UNIX and Windows servers to wait indefinitely for a lock or to wait for the specified number of seconds for a lock when the lock cannot be obtained immediately. The data type of this property is int. A value of zero means no wait. A value of -1 means to wait indefinitely. A positive integer indicates the number of seconds to wait for a lock.

currentPackagePath
Specifies a comma-separated list of collections on the server. The DB2 server searches these collections for the DB2 packages for the DB2 Universal JDBC Driver.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the DB2 CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

currentPackageSet
Specifies the collection ID to search for DB2 packages for the DB2 Universal JDBC Driver. The data type of this property is String. The default is NULLID. If currentPackageSet is set, its value overrides the value of jdbcCollection.

Multiple instances of the DB2 Universal JDBC Driver can be installed at a database server by running the DB2binder utility multiple times. The DB2binder utility includes a -collection option that lets the installer specify the collection ID for each DB2 Universal JDBC Driver instance. To choose an instance of the DB2 Universal JDBC Driver for a connection, you specify a currentPackageSet value that matches the collection ID for one of the DB2 Universal JDBC Driver instances.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the DB2 CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

currentSchema
Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements. This value of this property sets the value in the CURRENT SCHEMA special register on a server other than a DB2 UDB for z/OS server. Do not set this property for a DB2 UDB for z/OS server.

currentSQLID
Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 UDB for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

cursorSensitivity
Specifies whether the `java.sql.ResultSet.TYPE_SCROLL_SENSITIVE` value for a JDBC `ResultSet` maps to the SENSITIVE DYNAMIC attribute or the SENSITIVE STATIC attribute for the underlying DB2 cursor. Possible values are `TYPE_SCROLL_SENSITIVE_STATIC` and `TYPE_SCROLL_SENSITIVE_DYNAMIC`. The default is `TYPE_SCROLL_SENSITIVE_STATIC`.

This property is ignored for database servers that do not support sensitive dynamic scrollable cursors.

databaseName
Specifies the name for the database server. This name is used as the *database* portion of the connection URL. The name depends on whether Universal Type 4 Connectivity or Universal Type 2 Connectivity is used.

For Universal Type 4 Connectivity:

- If the connection is to a DB2 for z/OS server, the databaseName value is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

  ```
  SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
  ```

- If the connection is to a DB2 UDB for Linux, UNIX and Windows server, the databaseName value is the database name that is defined during installation.
- If the connection is to an IBM Cloudscape server, the databaseName value is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

  ```
  "c:/databases/testdb"
  ```

If this property is not set, connections are made to the local site.

For Universal Type 2 Connectivity:

- The databaseName value is the database name that is defined during installation, if the value of the serverName connection property is null. If the value of serverName property is not null, the databaseName value is a database alias.

deferPrepares
Specifies whether to defer prepare operations until run time. The data type of this property is boolean. The default is `true` for Universal Type 4 Connectivity. The property is not applicable to Universal Type 2 Connectivity.

Deferring prepare operations can reduce network delays. However, if you defer prepare operations, you need to ensure that input data types match DB2 table column types.

description
A description of the data source. The data type of this property is String.

driverType
For the `DataSource` interface, determines which driver to use for connections. The data type of this property is int. Valid values are 2 or 4. 2 is the default.

fullyMaterializeLobData
Indicates whether the driver retrieves LOB locators for FETCH operations. The data type of this property is boolean. If the value is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If this value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.

This property has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOB locators are always used for data that is fetched using scrollable cursors.

gssCredential
For a data source that uses Kerberos security, specifies a delegated credential that is passed from another principal. The data type of this property is org.ietf.jgss.GSSCredential. Delegated credentials are used in multi-tier environments, such as when a client connects to WebSphere Application Server, which, in turn, connects to DB2. You obtain a value for this property from the client, by invoking the `GSSContext.getDelegCred` method. `GSSContext` is part of the IBM Java Generic Security Service (GSS) API. If you set this property, you also need to set the Mechanism and KerberosServerPrincipal properties.

This property is applicable only to Universal Type 4 Connectivity.

For more information on using Kerberos security with the DB2 Universal JDBC Driver, see Using Kerberos security under the DB2 Universal JDBC Driver.

jdbcCollection
Specifies the collection ID for the packages that are used by an instance of the DB2 Universal JDBC Driver at run time. The data type of jdbcCollection is String. The default is NULLID.

This property is used with the DB2Binder -collection option. The DB2Binder utility must have previously bound DB2 Universal JDBC Driver packages at the server using a -collection value that matches the jdbcCollection value.

The jdbcCollection setting does not determine the collection that is used for SQLJ applications, For SQLJ, the collection is determined by the -collection option of the SQLJ customizer.

jdbcCollection does not apply to Universal Type 2 Connectivity on DB2 UDB for z/OS.

kerberosServerPrincipal
For a data source that uses Kerberos security, specifies the name that is used for the data source when it is registered with the Kerberos Key Distribution Center (KDC). The data type of this property is String.

This property is applicable only to Universal Type 4 Connectivity.

loginTimeout
The maximum time in seconds to wait for a connection to a data source, or for SQL requests to that data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The data type of this property is int. The default is 0. A value of 0 means that the timeout value is the default system timeout value. This property is not supported for Universal Type 2 Connectivity on DB2 UDB in the z/OS or OS/390 environment.

logWriter
The character output stream to which all logging and trace messages for the `DataSource` object are printed. The data type of this property is `java.io.PrinterWriter`. The default value is null, which means that no logging or tracing for the `DataSource` is output.

password
The password to use for establishing connections. The data type of this property is String. When you use the `DataSource` interface to establish a connection, you can override this property value by invoking this form of the `DataSource.getConnection` method:

```
getConnection(user, password);
```

portNumber
The port number where the DRDA® server is listening for requests. The data type of this property is int.

readOnly
Specifies whether the connection is read-only. The data type of this property is boolean. The default is `false`.

resultSetHoldability
Specifies whether cursors remain open after a commit operation. The data type of this property is int. Valid values are `com.ibm.db2.jcc.DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT` or `com.ibm.db2.jcc.DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT`. These values are the same as the `ResultSet.HOLD_CURSORS_OVER_COMMIT` and `ResultSet.CLOSE_CURSORS_AT_COMMIT` constants that are defined in JDBC 3.0.

retrieveMessagesFromServerOnGetMessage
Specifies whether JDBC `SQLException.getMessage` calls cause the DB2 Universal JDBC Driver to invoke a DB2 UDB for OS/390 or z/OS stored procedure that retrieves the message text for the error. The data type of this property is boolean. The default is `false`, which means that the full message text is not returned to the client. An alternative to setting this property to `true` is to use the DB2-only `DB2Sqlca.getMessage` method in applications. Both techniques result in a stored procedure call, which starts a unit of work.

securityMechanism
Specifies the DRDA security mechanism. The data type of this property in int. Possible values are:

CLEAR_TEXT_PASSWORD_SECURITY
User ID and password

USER_ONLY_SECURITY
User ID only

ENCRYPTED_PASSWORD_SECURITY
User ID, encrypted password

ENCRYPTED_USER_AND_PASSWORD_SECURITY
Encrypted user ID and password

KERBEROS_SECURITY
Kerberos

If this property is specified, the specified security mechanism is the only mechanism that is used. If the security mechanism is not supported by the connection, an exception is thrown.

If no value is specified for this property, the requester attempts to connect using the most secure mechanism that is possible. If a connection cannot be established because the server does not support that security mechanism, the server returns a list of alternate choices to the requester. The requester tries each of those security mechanisms until a connection can be established with one of them. If there are no alternative choices, or if all alternative choices fail, an exception is thrown.

serverName
The host name or the TCP/IP address of the data source. The data type of this property is String.

traceFile
Specifies the name of a file into which the DB2 Universal JDBC Driver writes trace information. The data type of this property is `String`. The traceFile property is an alternative to the logWriter property for directing the output trace stream to a file.

traceFileAppend
Specifies whether to append to or overwrite the file that is specified by the traceFile property. The data type of this property is boolean. The default is `false`, which means that the file that is specified by the traceFile property is overwritten.

traceLevel
Specifies what to trace. The data type of this property is int.

You can specify one or more of the following traces with the traceLevel property:

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (Universal Type 2 Connectivity for DB2 UDB for Linux, UNIX and Windows only)`

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL`

To specify more than one trace, use one of these techniques:

- Use bitwise OR (`|`) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:
  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```
- Use a bitwise complement (~) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:
  ```
  ~TRACE_DRDA_FLOWS
  ```

user

The user ID to use for establishing connections. The data type of this property is String. When you use the `DataSource` interface to establish a connection, you can override this property value by invoking this form of the `DataSource.getConnection` method:

```
getConnection(user, password);
```

Related concepts:

- "Security under the DB2 Universal JDBC Driver" on page 444
- "LOBs in JDBC applications with the DB2 Universal JDBC Driver" on page 289

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426
- "Summary of DB2 Universal JDBC Driver extensions to JDBC" on page 414

Comparison of driver support for JDBC APIs

The following tables list the JDBC interfaces and indicate which drivers supports them. The drivers and their supported platforms are:

Table 44. JDBC drivers for DB2 UDB

JDBC driver name	Associated DB2 UDB
DB2 Universal JDBC Driver	DB2 UDB for Linux, UNIX and Windows or DB2 UDB for z/OS
JDBC/SQLJ 2.0 Driver for OS/390	DB2 UDB for z/OS
DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (deprecated)	DB2 UDB for Linux, UNIX and Windows

Table 45. DB2 JDBC support for Array methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getArray	No	No	No
getBaseType	No	No	No

Table 45. DB2 JDBC support for Array methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getBaseTypeName	No	No	No
getResultSet	No	No	No

Table 46. DB2 JDBC support for BatchUpdateException methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getUpdateCounts	Yes	Yes	Yes

Table 47. DB2 JDBC support for Blob methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getBinaryStream	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setBinaryStream[1]	Yes	No	No
setBytes[1]	Yes	No	No
truncate[1]	Yes	No	No

Notes:

1. This is a JDBC 3.0 method.

Table 48. DB2 JDBC support for CallableStatement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
Methods inherited from java.sql.PreparedStatement	Yes	Yes	Yes
getArray	No	No	No
getBigDecimal	Yes	Yes	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getDate	Yes	Yes	Yes
getDouble	Yes	Yes	Yes

Table 48. DB2 JDBC support for CallableStatement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getObject	Yes[1]	Yes[1]	Yes[1]
getRef	No	No	No
getShort	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes	Yes	Yes
getTimestamp	Yes	Yes	Yes
registerOutParameter[2]	Yes	Yes	Yes
wasNull	Yes	Yes	Yes

Notes:

1. The following form of the `getObject` method is *not* supported:
   `getObject(int parameterIndex, java.util.Map map)`
2. The following form of the `registerOutParameter` method is *not* supported:
   `registerOutParameter(int parameterIndex, int jdbcType, String typeName)`

Table 49. DB2 JDBC support for Clob methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getAsciiStream	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getSubString	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setAsciiStream[1]	Yes	No	No
setCharacterStream[1]	Yes	No	No
setString[1]	Yes	No	No
truncate[1]	Yes	No	No

Notes:

1. This is a JDBC 3.0 method.

Table 50. DB2 JDBC support for Connection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes

Table 50. DB2 JDBC support for Connection methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
commit	Yes	Yes	Yes
createStatement	Yes[1]	Yes [2]	Yes
getAutoCommit	Yes	Yes	Yes
getCatalog	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getTransactionIsolation	Yes	Yes	Yes
getTypeMap	No	No	No
getWarnings	Yes	Yes	Yes
isClosed	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
nativeSQL	Yes	Yes	Yes
prepareCall	Yes	Yes[3]	Yes
prepareStatement	Yes[4]	Yes	Yes
releaseSavepoint	Yes[5]	No	No
rollback	Yes	Yes[6]	Yes[6]
setAutoCommit	Yes	Yes	Yes
setCatalog	Yes	Yes	Yes
setReadOnly	Yes[7]	Yes[7]	Yes
setSavepoint	Yes[5]	No	No
setTransactionIsolation	Yes	Yes	Yes
setTypeMap	No	No	No

Notes:

1. In addition to the JDBC 2.0 forms of `createStatement` statement, the following JDBC 3.0 form of `createStatement` is supported:

   ```
   createStatement(int resultSetType,
    int resultSetConcurrency,
    int resultSetHoldability)
   ```

2. For the following form of `createStatement`, a *resultSetType* value of `TYPE_FORWARD_ONLY` and a *resultSetConcurrency* value of `CONCUR_READ_ONLY` are supported:

   ```
   createStatement(int resultSetType, int resultSetConcurrency)
   ```

3. The following form of `prepareCall` is *not* supported:

   ```
   prepareCall(String sql, int resultSetType, int resultSetConcurrency)
   ```

4. In addition to the other forms of `prepareStatement`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 form:

   ```
   prepareStatement(String sql, int autoGeneratedKeys)
   ```

5. This is a JDBC 3.0 method.
6. The JDBC 3.0 `rollback(Savepoint savepoint)` method is not supported.
7. The driver does not use the setting. For the DB2 Universal JDBC Driver, a connection can be set as read-only through the `readOnly` property for a `Connection` or `DataSource` object.

Table 51. DB2 JDBC support for ConnectionEvent methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.util.EventObject	Yes	Yes	Yes
getSQLException	Yes	Yes	Yes

Table 52. DB2 JDBC support for ConnectionEventListener methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
connectionClosed	Yes	Yes	Yes
connectionErrorOccurred	Yes	Yes	Yes

Table 53. DB2 JDBC support for ConnectionPoolDataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
getPooledConnection	Yes	Yes	Yes
setLoginTimeout	Yes[1]	Yes	Yes
setLogWriter	Yes	Yes	Yes

Note:

1. This method is not supported for Universal Type 2 Connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 54. DB2 JDBC support for DatabaseMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
allProceduresAreCallable	Yes	Yes	Yes
allTablesAreSelectable	Yes	Yes	Yes
dataDefinitionCausesTransactionCommit	Yes	Yes	Yes
dataDefinitionIgnoredInTransactions	Yes	Yes	Yes
deletesAreDetected	Yes	Yes	Yes
doesMaxRowSizeIncludeBlobs	Yes	Yes	Yes
getAttributes	Yes	No	No
getBestRowIdentifier	Yes	Yes	Yes
getCatalogs	Yes	Yes	Yes
getCatalogSeparator	Yes	Yes	Yes
getCatalogTerm	Yes	Yes	Yes
getColumnPrivileges	Yes	Yes	Yes

Table 54. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getColumns	Yes[1]	Yes	Yes
getConnection	Yes	Yes	Yes
getCrossReference	Yes	Yes	Yes
getDatabaseMajorVersion	Yes	No	No
getDatabaseMinorVersion	Yes	No	No
getDatabaseProductName	Yes	Yes	Yes
getDatabaseProductVersion	Yes	Yes	Yes
getDefaultTransactionIsolation	Yes	Yes	Yes
getDriverMajorVersion	Yes	Yes	Yes
getDriverMinorVersion	Yes	Yes	Yes
getDriverName	Yes	Yes	Yes
getDriverVersion	Yes	Yes	Yes
getExportedKeys	Yes	Yes	Yes
getExtraNameCharacters	Yes	Yes	Yes
getIdentifierQuoteString	Yes	Yes	Yes
getImportedKeys	Yes	Yes	Yes
getIndexInfo	Yes	Yes	Yes
getJDBCMajorVersion	Yes	No	No
getJDBCMinorVersion	Yes	No	No
getMaxBinaryLiteralLength	Yes	Yes	Yes
getMaxCatalogNameLength	Yes	Yes	Yes
getMaxCharLiteralLength	Yes	Yes	Yes
getMaxColumnNameLength	Yes	Yes	Yes
getMaxColumnsInGroupBy	Yes	Yes	Yes
getMaxColumnsInIndex	Yes	Yes	Yes
getMaxColumnsInOrderBy	Yes	Yes	Yes
getMaxColumnsInSelect	Yes	Yes	Yes
getMaxColumnsInTable	Yes	Yes	Yes
getMaxConnections	Yes	Yes	Yes
getMaxCursorNameLength	Yes	Yes	Yes
getMaxIndexLength	Yes	Yes	Yes
getMaxProcedureNameLength	Yes	Yes	Yes
getMaxRowSize	Yes	Yes	Yes
getMaxSchemaNameLength	Yes	Yes	Yes
getMaxStatementLength	Yes	Yes	Yes
getMaxStatements	Yes	Yes	Yes

Table 54. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getMaxTableNameLength	Yes	Yes	Yes
getMaxTablesInSelect	Yes	Yes	Yes
getMaxUserNameLength	Yes	Yes	Yes
getNumericFunctions	Yes	Yes	Yes
getPrimaryKeys	Yes	Yes	Yes
getProcedureColumns	Yes	Yes	Yes
getProcedures	Yes	Yes	Yes
getProcedureTerm	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getSchemas	Yes[1]	Yes	Yes
getSchemaTerm	Yes	Yes	Yes
getSearchStringEscape	Yes	Yes	Yes
getSQLKeywords	Yes	Yes	Yes
getSQLStateType	Yes	No	No
getStringFunctions	Yes	Yes	Yes
getSuperTables	Yes[2]	No	No
getSuperTypes	Yes[2]	No	No
getSystemFunctions	Yes	Yes	Yes
getTablePrivileges	Yes	Yes	Yes
getTables	Yes[1]	Yes	Yes
getTableTypes	Yes	Yes	Yes
getTimeDateFunctions	Yes	Yes	Yes
getTypeInfo	Yes	Yes	Yes
getUDTs	No	No	Yes[2]
getURL	Yes	Yes	Yes
getUserName	Yes	Yes	Yes
getVersionColumns	Yes	Yes	Yes
insertsAreDetected	Yes	Yes	Yes
isCatalogAtStart	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
nullPlusNonNullIsNull	Yes	Yes	Yes
nullsAreSortedAtEnd	Yes	Yes	Yes
nullsAreSortedAtStart	Yes	Yes	Yes
nullsAreSortedHigh	Yes	Yes	Yes
nullsAreSortedLow	Yes	Yes	Yes
othersDeletesAreVisible	Yes	Yes	Yes

Table 54. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
othersInsertsAreVisible	Yes	Yes	Yes
othersUpdatesAreVisible	Yes	Yes	Yes
ownDeletesAreVisible	Yes	Yes	Yes
ownInsertsAreVisible	Yes	Yes	Yes
ownUpdatesAreVisible	Yes	Yes	Yes
storesLowerCaseIdentifiers	Yes	Yes	Yes
storesLowerCaseQuotedIdentifiers	Yes	Yes	Yes
storesMixedCaseIdentifiers	Yes	Yes	Yes
storesMixedCaseQuotedIdentifiers	Yes	Yes	Yes
storesUpperCaseIdentifiers	Yes	Yes	Yes
storesUpperCaseQuotedIdentifiers	Yes	Yes	Yes
supportsAlterTableWithAddColumn	Yes	Yes	Yes
supportsAlterTableWithDropColumn	Yes	Yes	Yes
supportsANSI92EntryLevelSQL	Yes	Yes	Yes
supportsANSI92FullSQL	Yes	Yes	Yes
supportsANSI92IntermediateSQL	Yes	Yes	Yes
supportsBatchUpdates	Yes	Yes	Yes
supportsCatalogsInDataManipulation	Yes	Yes	Yes
supportsCatalogsInIndexDefinitions	Yes	Yes	Yes
supportsCatalogsInPrivilegeDefinitions	Yes	Yes	Yes
supportsCatalogsInProcedureCalls	Yes	Yes	Yes
supportsCatalogsInTableDefinitions	Yes	Yes	Yes
SupportsColumnAliasing	Yes	Yes	Yes
supportsConvert	Yes	Yes	Yes
supportsCoreSQLGrammar	Yes	Yes	Yes
supportsCorrelatedSubqueries	Yes	Yes	Yes
supportsDataDefinitionAndDataManipulationTransactions	Yes	Yes	Yes
supportsDataManipulationTransactionsOnly	Yes	Yes	Yes
supportsDifferentTableCorrelationNames	Yes	Yes	Yes
supportsExpressionsInOrderBy	Yes	Yes	Yes
supportsExtendedSQLGrammar	Yes	Yes	Yes
supportsFullOuterJoins	Yes	Yes	Yes
supportsGetGeneratedKeys	Yes	No	No
supportsGroupBy	Yes	Yes	Yes
supportsGroupByBeyondSelect	Yes	Yes	Yes
supportsGroupByUnrelated	Yes	Yes	Yes

Table 54. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsIntegrityEnhancementFacility	Yes	Yes	Yes
supportsLikeEscapeClause	Yes	Yes	Yes
supportsLimitedOuterJoins	Yes	Yes	Yes
supportsMinimumSQLGrammar	Yes	Yes	Yes
supportsMixedCaseIdentifiers	Yes	Yes	Yes
supportsMixedCaseQuotedIdentifiers	Yes	Yes	Yes
supportsMultipleOpenResults	Yes	Yes	No
supportsMultipleResultSets	Yes	Yes	Yes
supportsMultipleTransactions	Yes	Yes	Yes
supportsNamedParameters	Yes	No	No
supportsNonNullableColumns	Yes	Yes	Yes
supportsOpenCursorsAcross Commit	Yes	Yes	Yes
supportsOpenCursorsAcross Rollback	Yes	Yes	Yes
supportsOpenStatementsAcrossCommit	Yes	Yes	Yes
supportsOpenStatementsAcrossRollback	Yes	Yes	Yes
supportsOrderByUnrelated	Yes	Yes	Yes
supportsOuterJoins	Yes	Yes	Yes
supportsPositionedDelete	Yes	Yes	Yes
supportsPositionedUpdate	Yes	Yes	Yes
supportsResultSetConcurrency	Yes	Yes	Yes
supportsResultSetHoldability	Yes	No	No
supportsResultSetType	Yes	Yes	Yes
supportsSavepoints	Yes	No	No
supportsSchemasInDataManipulation	Yes	Yes	Yes
supportsSchemasInIndexDefinitions	Yes	Yes	Yes
supportsSchemasInPrivilegeDefinitions	Yes	Yes	Yes
supportsSchemasInProcedureCalls	Yes	Yes	Yes
supportsSchemasInTableDefinitions	Yes	Yes	Yes
supportsSelectForUpdate	Yes	Yes	Yes
supportsStoredProcedures	Yes	Yes	Yes
supportsSubqueriesInComparisons	Yes	Yes	Yes
supportsSubqueriesInExists	Yes	Yes	Yes
supportsSubqueriesInIns	Yes	Yes	Yes
supportsSubqueriesInQuantifieds	Yes	Yes	Yes
supportsSuperTables	Yes	No	No
supportsSuperTypes	Yes	No	No

Table 54. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsTableCorrelationNames	Yes	Yes	Yes
supportsTransactionIsolationLevel	Yes	Yes	Yes
supportsTransactions	Yes	Yes	Yes
supportsUnion	Yes	Yes	Yes
supportsUnionAll	Yes	Yes	Yes
updatesAreDetected	Yes	Yes	Yes
usesLocalFilePerTable	Yes	Yes	Yes
usesLocalFiles	Yes	Yes	Yes

Notes:

1. The JDBC 3.0 version of this method is supported.
2. The method can be executed, but it returns an empty `ResultSet`.

Table 55. DB2 JDBC support for DataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getConnection	Yes	Yes	Yes
getLoginTimeout	Yes	Yes	Yes[1]
getLogWriter	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes	Yes[1]
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for Universal Type 2 Connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 56. DB2 JDBC support for DataTruncation methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Throwable	Yes	Yes	Yes
Methods inherited from java.sql.SQLException	Yes	Yes	Yes
Methods inherited from java.sql.SQLWarning	Yes	Yes	Yes
getDataSize	Yes	Yes	Yes
getIndex	Yes	Yes	Yes
getParameter	Yes	Yes	Yes
getRead	Yes	Yes	Yes

Table 56. DB2 JDBC support for DataTruncation methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getTransferSize	Yes	Yes	Yes

Table 57. DB2 JDBC support for Driver methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
acceptsURL	Yes	Yes	Yes
connect	Yes	Yes	Yes
getMajorVersion	Yes	Yes	Yes
getMinorVersion	Yes	Yes	Yes
getPropertyInfo	Yes	Yes	Yes
jdbcCompliant	Yes	Yes	Yes

Table 58. DB2 JDBC support for DriverManager methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
deregisterDriver	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
getDriver	Yes	Yes	Yes
getDrivers	Yes	Yes	Yes
getLoginTimeout	Yes	Yes	Yes[1]
getLogStream	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
println	Yes	Yes	Yes
registerDriver	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes	Yes[1]
setLogStream	Yes	Yes	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for Universal Type 2 Connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 59. DB2 JDBC support for ParameterMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterClassName	No	No	No
getParameterCount	Yes	No	No
getParameterMode	Yes	No	No

Table 59. DB2 JDBC support for ParameterMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterType	Yes	No	No
getParameterTypeName	Yes	No	No
getPrecision	Yes	No	No
getScale	Yes	No	No
isNullable	Yes	No	No
isSigned	Yes	No	No

Table 60. DB2 JDBC support for PooledConnection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addConnectionEventListener	Yes	Yes	Yes
close	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
removeConnectionEventListener	Yes	Yes	Yes

Table 61. DB2 JDBC support for PreparedStatement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
addBatch	Yes	Yes	Yes
clearParameters	Yes	Yes	Yes
execute	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
setArray	No	No	No
setAsciiStream	Yes	Yes	Yes
setBigDecimal	Yes	Yes	Yes
setBinaryStream	Yes	Yes	Yes
setBlob	Yes	Yes	Yes
setBoolean	Yes	Yes	Yes
setByte	Yes	Yes	Yes
setBytes	Yes	Yes	Yes
setCharacterStream	Yes	Yes	Yes
setClob	Yes	Yes	Yes
setDate	Yes	Yes[1]	Yes
setDouble	Yes	Yes	Yes

Table 61. DB2 JDBC support for PreparedStatement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
setFloat	Yes	Yes	Yes
setInt	Yes	Yes	Yes
setLong	Yes	Yes	Yes
setNull	Yes[2]	Yes[2]	Yes[2]
setObject	Yes	Yes	Yes
setRef	No	No	No
setShort	Yes	Yes	Yes
setString	Yes[3]	Yes[3]	Yes[3]
setTime	Yes[4]	Yes[4]	Yes
setTimestamp	Yes[5]	Yes[5]	Yes
setUnicodeStream	Yes	Yes	Yes
setURL	Yes	No	Yes

Notes:

1. The following form of `setDate` is *not* supported:
   `setDate(int parameterIndex, java.sql.Date x, java.util.Calendar cal)`
2. The following form of `setNull` is *not* supported:
   `setNull(int parameterIndex, int jdbcType, String typeName)`
3. setString is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.
4. The following form of `setTime` is *not* supported:
   `setTime(int parameterIndex, java.sql.Time x, java.util.Calendar cal)`
5. The following form of `setTimestamp` is *not* supported:
   `setTimestamp(int parameterIndex, java.sql.Timestamp x, java.util.Calendar cal)`

Table 62. DB2 JDBC support for Ref methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
get BaseTypeName	No	No	No

Table 63. DB2 JDBC support for ResultSet methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
absolute	Yes	No	Yes
afterLast	Yes	No	Yes
beforeFirst	Yes	No	Yes
cancelRowUpdates	Yes	No	No
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
deleteRow	Yes	No	No

Table 63. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
findColumn	Yes	Yes	Yes
first	Yes	No	Yes
getArray	No	No	No
getAsciiStream	Yes	Yes	Yes
getBigDecimal	Yes	Yes	Yes
getBinaryStream	Yes[1]	Yes[1]	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getConcurrency	Yes	Yes	Yes
getCursorName	Yes	Yes	Yes
getDate	Yes	Yes[2]	Yes
getDouble	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getObject	Yes[3]	Yes[3]	Yes[3]
getRef	No	No	No
getRow	Yes	No	Yes
getShort	Yes	Yes	Yes
getStatement	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes	Yes[4]	Yes
getTimestamp	Yes	Yes[5]	Yes
getType	Yes	Yes	Yes
getUnicodeStream	Yes	Yes	Yes
getURL	Yes	No	Yes
getWarnings	Yes	Yes	Yes
insertRow	No	No	No
isAfterLast	Yes	No	Yes
isBeforeFirst	Yes	No	Yes
isFirst	Yes	No	Yes

Table 63. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
isLast	Yes	No	Yes
last	Yes	No	Yes
moveToCurrentRow	Yes	No	No
moveToInsertRow	No	No	No
next	Yes	Yes	Yes
previous	Yes	No	Yes
refreshRow	Yes	No	No
relative	Yes	No	Yes
rowDeleted	Yes	No	No
rowInserted	No	No	No
rowUpdated	Yes	No	No
setFetchDirection	Yes	Yes[6]	Yes
setFetchSize	Yes	Yes	Yes
updateAsciiStream	Yes	No	No
updateBigDecimal	Yes	No	No
updateBinaryStream	Yes	No	No
updateBoolean	Yes	No	No
updateByte	Yes	No	No
updateBytes	Yes	No	No
updateCharacterStream	Yes	No	No
updateDate	Yes	No	No
updateDouble	Yes	No	No
updateFloat	Yes	No	No
updateInt	Yes	No	No
updateLong	Yes	No	No
updateNull	Yes	No	No
updateObject	Yes	No	No
updateRow	Yes	No	No
updateShort	Yes	No	No
updateString	Yes	No	No
updateTime	Yes	No	No
updateTimestamp	Yes	No	No
wasNull	Yes	Yes	Yes

Table 63. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support

Notes:

1. `getBinaryStream` is not supported for CLOB columns.
2. The following forms of `getDate` are *not* supported:

   ```
   getDate(int columnIndex, java.util.Calendar cal)
   getDate(String columnName, java.util.Calendar cal)
   ```

3. The following form of the `getObject` method is *not* supported:

   ```
   getObject(int parameterIndex, java.util.Map  map)
   ```

4. The following forms of `getTime` are *not* supported:

   ```
   getTime(int columnIndex, java.util.Calendar cal)
   getTime(String columnName, java.util.Calendar cal)
   ```

5. The following forms of `getTimestamp` are *not* supported:

   ```
   getTimestamp(int columnIndex, java.util.Calendar cal)
   getTimestamp(String columnName, java.util.Calendar cal)
   ```

6. Supported only if *direction* is `ResultSet.FETCH_FORWARD`.

Table 64. DB2 JDBC support for ResultSetMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getCatalogName	Yes	Yes	Yes
getColumnClassName	No	No	Yes
getColumnCount	Yes	Yes	Yes
getColumnDisplaySize	Yes	Yes	Yes
getColumnLabel	Yes	Yes	Yes
getColumnName	Yes	Yes	Yes
getColumnType	Yes	Yes	Yes
getColumnTypeName	Yes	Yes	Yes
getPrecision	Yes	Yes	Yes
getScale	Yes	Yes	Yes
getSchemaName	Yes	Yes	Yes
getTableName	Yes	Yes	Yes
isAutoIncrement	Yes	Yes	Yes
isCaseSensitive	Yes	Yes	Yes
isCurrency	Yes	Yes	Yes
isDefinitelyWritable	Yes	Yes	Yes
isNullable	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
isSearchable	Yes	Yes	Yes
isSigned	Yes	Yes	Yes
isWritable	Yes	Yes	Yes

Table 65. DB2 JDBC support for SQLData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No	No
readSQL	No	No	No
writeSQL	No	No	No

Table 66. DB2 JDBC support for SQLException methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getSQLState	Yes	Yes	Yes
getErrorCode	Yes	Yes	Yes
getNextException	Yes	Yes	Yes
setNextException	Yes	Yes	Yes

Table 67. DB2 JDBC support for SQLInput methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
readArray	No	No	No
readAsciiStream	No	No	No
readBigDecimal	No	No	No
readBinaryStream	No	No	No
readBlob	No	No	No
readBoolean	No	No	No
readByte	No	No	No
readBytes	No	No	No
readCharacterStream	No	No	No
readClob	No	No	No
readDate	No	No	No
readDouble	No	No	No
readFloat	No	No	No
readInt	No	No	No
readLong	No	No	No
readObject	No	No	No
readRef	No	No	No
readShort	No	No	No
readString	No	No	No
readTime	No	No	No
readTimestamp	No	No	No

Table 67. DB2 JDBC support for SQLInput methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
wasNull	No	No	No

Table 68. DB2 JDBC support for SQLOutput methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
writeArray	No	No	No
writeAsciiStream	No	No	No
writeBigDecimal	No	No	No
writeBinaryStream	No	No	No
writeBlob	No	No	No
writeBoolean	No	No	No
writeByte	No	No	No
writeBytes	No	No	No
writeCharacterStream	No	No	No
writeClob	No	No	No
writeDate	No	No	No
writeDouble	No	No	No
writeFloat	No	No	No
writeInt	No	No	No
writeLong	No	No	No
writeObject	No	No	No
writeRef	No	No	No
writeShort	No	No	No
writeString	No	No	No
writeStruct	No	No	No
writeTime	No	No	No
writeTimestamp	No	No	No

Table 69. DB2 JDBC support for Statement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addBatch	Yes	Yes	Yes
cancel	Yes[1]	No	Yes
clearBatch	Yes	Yes	Yes
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
execute	Yes[2]	Yes	Yes
executeBatch	Yes	Yes	Yes

Table 69. DB2 JDBC support for Statement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
executeQuery	Yes	Yes	Yes
executeUpdate	Yes[2]	Yes	Yes
getConnection	Yes	No	Yes
getFetchDirection	Yes	No	Yes
getFetchSize	Yes	No	Yes
getGeneratedKeys	Yes	No	No
getMaxFieldSize	Yes	Yes	Yes
getMaxRows	Yes	Yes	Yes
getMoreResults	Yes[3]	Yes	Yes
getQueryTimeout	Yes[1]	Yes	Yes
getResultSet	Yes	Yes	Yes
getResultSetConcurrency	Yes	Yes	Yes
getResultSetType	Yes	Yes	Yes
getUpdateCount[4]	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
setCursorName	Yes	Yes	Yes
setEscapeProcessing	Yes	Yes	Yes
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	No	Yes
setMaxFieldSize	Yes	Yes	Yes
setMaxRows	Yes	Yes	Yes
setQueryTimeout	Yes[5]	Yes[5]	Yes

Notes:

1. This method is not supported for Universal Type 2 Connectivity in the OS/390 or z/OS environment.
2. In addition to the other forms of `execute` or `executeUpdate`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 forms:

   ```
   executeUpdate(String sql, int autoGeneratedKeys)
   execute(String sql, int autoGeneratedKeys)
   ```

3. In addition to `getMoreResults()`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 forms:
   - `getMoreResults(java.sql.Statement.CLOSE_CURRENT_RESULT)`
   - `getMoreResults(java.sql.Statement.KEEP_CURRENT_RESULT)`
   - `getMoreResults(java.sql.Statement.CLOSE_ALL_RESULTS)`
4. Not supported for stored procedure `ResultSets`.
5. Supported only for a *seconds* value of 0.

Table 70. DB2 JDBC support for Struct methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No	No
getAttributes	No	No	No

Table 71. DB2 JDBC support for XAConnection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from javax.sql.PooledConnection	Yes[1]	No	Yes
getXAResource	Yes[1]	No	Yes

Notes:

1. This method is supported for DB2 Universal JDBC Driver type 2 connectivity to a DB2 UDB for Linux, UNIX and Windows server or DB2 Universal JDBC Driver type 4 connectivity to a DB2 UDB for z/OS server.

Table 72. DB2 JDBC support for XADataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	No	Yes
getLogWriter	Yes	No	Yes
getXAConnection	Yes	No	Yes
setLoginTimeout	Yes	No	Yes
setLogWriter	Yes	No	Yes

Related reference:

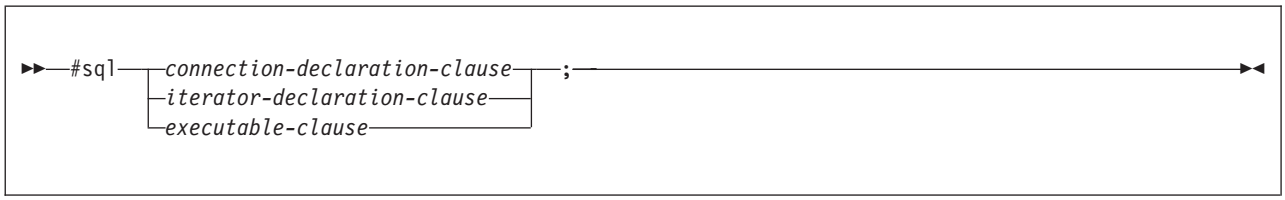
- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426

SQLJ statement reference

The sections that follow contain information about the syntax of SQLJ clauses.

SQLJ clause

The SQL statements in an SQLJ program are in SQLJ clauses. The general syntax of an SQLJ clause is:



Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

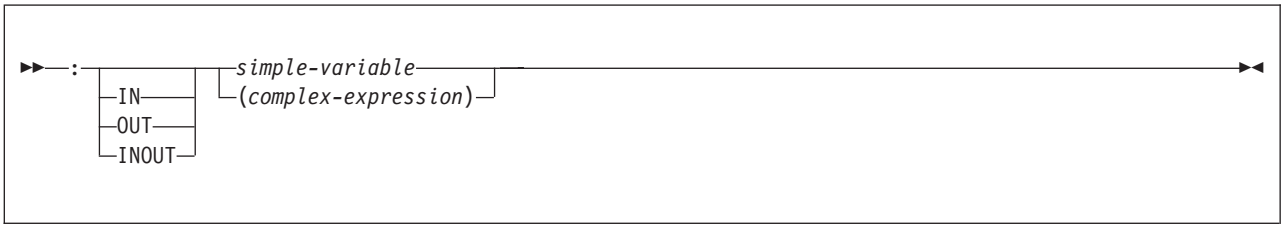
Related reference:

- "SQLJ connection-declaration-clause" on page 399
- "SQLJ executable-clause" on page 401
- "SQLJ iterator-declaration-clause" on page 400

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax:



Description:

: Indicates that the variable or expression that follows is a host expression. The colon must immediately precede the variable or expression.

IN | OUT | INOUT
For a host expression that is used as a parameter in a stored procedure call, identifies whether the parameter provides data to the stored procedure (IN), retrieves data from the stored procedure (OUT), or does both (INOUT). The default is IN.

simple-variable
Specifies a Java unqualified identifier.

complex-expression
Specifies a Java expression that results in a single value.

Usage notes:

- A complex expression must be enclosed in parentheses.
- ANSI/ISO rules govern where a host expression can appear in a static SQL statement.

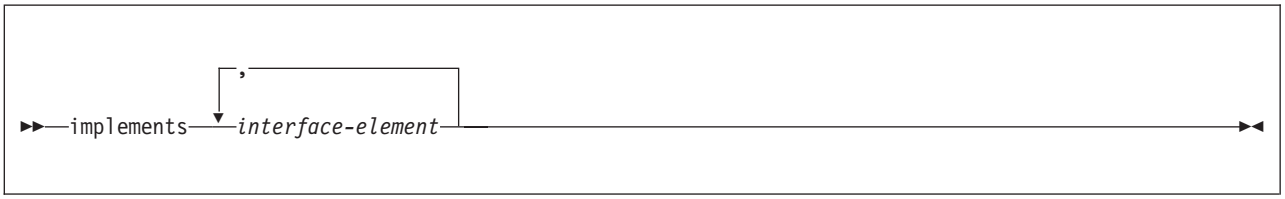
Related concepts:

- "Variables in SQLJ applications" on page 320

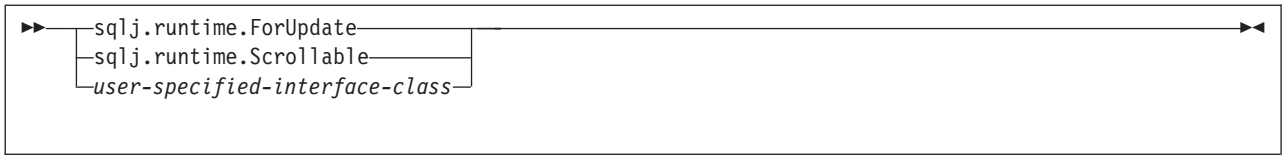
SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax:



interface-element:



Description:

interface-element
Specifies a user-defined Java interface, the SQLJ interface `sqlj.runtime.ForUpdate` or the SQLJ interface `sqlj.runtime.Scrollable`.

You need to implement `sqlj.runtime.ForUpdate` when you declare an iterator for a positioned UPDATE or positioned DELETE operation. See Perform positioned UPDATE and DELETE operations in an SQLJ application for information on performing a positioned UPDATE or positioned DELETE operation in SQLJ.

You need to implement `sqlj.runtime.Scrollable` when you declare a scrollable iterator. See Use scrollable iterators in an SQLJ application for information on scrollable iterators.

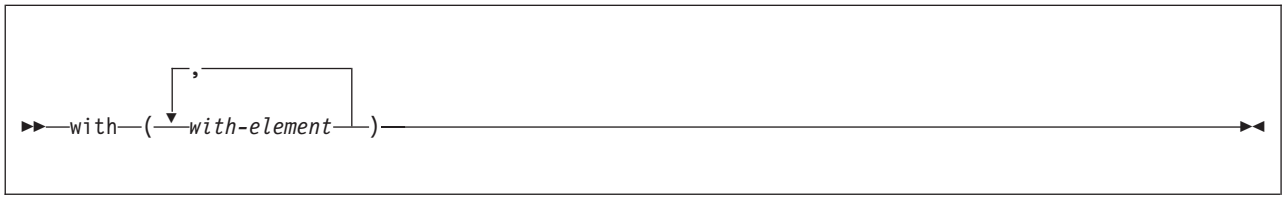
Related tasks:

- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336
- "Using scrollable iterators in an SQLJ application" on page 361

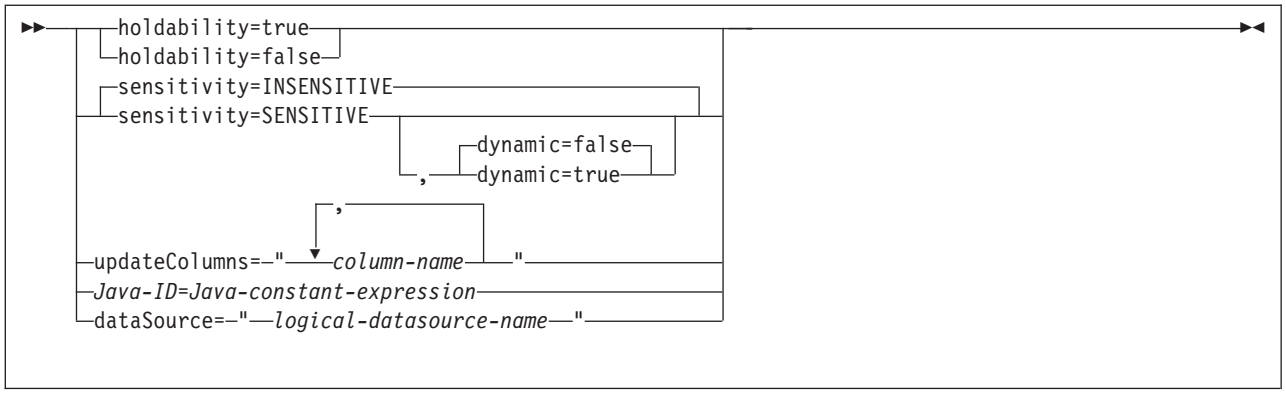
SQLJ with-clause

The with clause specifies a set of one or more attributes for an iterator or a connection context.

Syntax:



with-element:



Description:

holdability
For an iterator, specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be `true` or `false`.

sensitivity
For an iterator, specifies whether changes that are made to the underlying table can be visible to the iterator after it is opened. The value must be INSENSITIVE or SENSITIVE. The default is INSENSITIVE.

dynamic
For an iterator that is defined with sensitivity=SENSITIVE, specifies whether the following cases are true:

- When the application executes positioned UPDATE and DELETE statements with the iterator, those changes are visible to the iterator.
- When the application executes INSERT, UPDATE, and DELETE statements within the application but outside the iterator, those changes are visible to the iterator.

The value for dynamic must be `true` or `false`. The default is `false`.

If the value of dynamic is `true`, the data source must support dynamic scrollable cursors.

updateColumns
For an iterator, specifies the columns that are to be modified when the iterator is used for a positioned UPDATE statement. The value for updateColumns must be a literal string that contains the column names, separated by commas.

column-name
For an iterator, specifies a column of the result table that is to be updated using the iterator.

Java-ID
For an iterator or connection context, specifies a Java variable that identifies a user-defined attribute of the iterator or connection context. The value of *Java-constant-expression* is also user-defined.

dataSource
For a connection context, specifies the logical name of a separately-created `DataSource` object that represents the data source to which the application will connect. This option is available only for the DB2 Universal JDBC Driver.

Usage notes:

- The value on the left side of a with element must be unique within its with clause.

- If you specify updateColumns in a with element of an iterator declaration clause, the iterator declaration clause must also contain an implements clause that specifies the sqlj.runtime.ForUpdate interface.
- If you do not customize your SQLJ program, the JDBC driver ignores the value of holdability that is in the with clause. Instead, the driver uses the JDBC driver setting for holdability.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 345

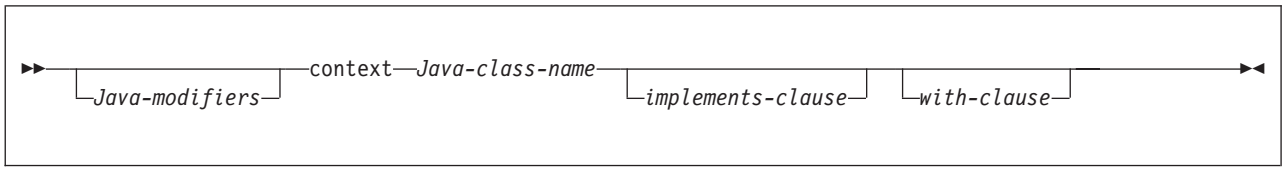
Related tasks:

- "Connecting to a data source using SQLJ" on page 322
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336
- "Using scrollable iterators in an SQLJ application" on page 361

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax:



Description:

Java-modifiers
Specifies modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Specifies a valid Java identifier. During the program preparation process, SQLJ generates a connection context class whose name is this identifier.

implements-clause
See SQLJ implements-clause for a description of this clause. In a connection declaration clause, the interface class to which the implements clause refers must be a user-defined interface class.

with-clause
See SQLJ with-clause for a description of this clause.

Usage notes:

- SQLJ generates a connection class declaration for each connection declaration clause you specify. SQLJ data source connections are objects of those generated connection classes.
- You can specify a connection declaration clause anywhere that a Java class definition can appear in a Java program.

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

Related reference:

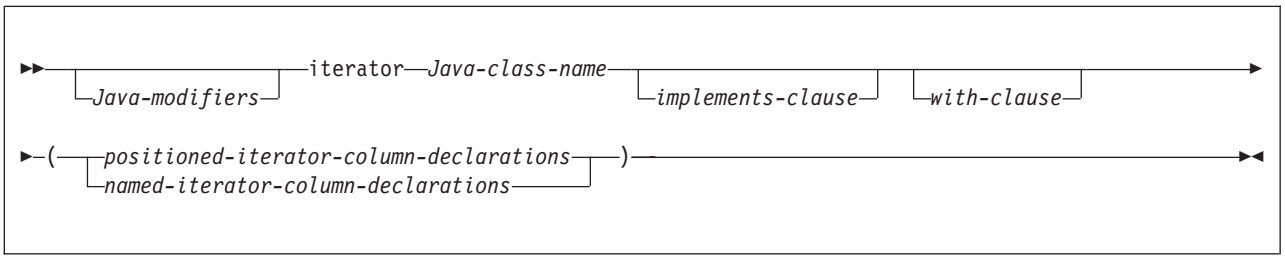
- "SQLJ implements-clause" on page 396
- "SQLJ with-clause" on page 397

SQLJ iterator-declaration-clause

An iterator declaration clause declares a positioned iterator class or a named iterator class in an SQLJ application program. An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator declaration clause you specify. An iterator is an object of an iterator class.

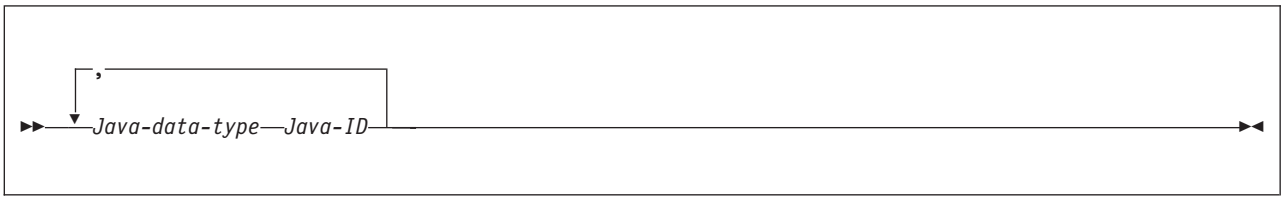
An iterator declaration clause has a form for a positioned iterator and a form for a named iterator. The two kinds of iterators are distinct and incompatible Java types that are implemented with different interfaces.

Syntax:



positioned-iterator-column declarations:

named-iterator-column-declarations:



Description:

Java-modifiers
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Any valid Java identifier. During the program preparation process, SQLJ generates an iterator class whose name is this identifier.

implements-clause
See SQLJ implements-clause for a description of this clause. For an iterator declaration clause that declares an iterator for a positioned UPDATE or

positioned DELETE operation, the implements clause must specify interface `sqlj.runtime.ForUpdate`. For an iterator declaration clause that declares a scrollable iterator, the implements clause must specify interface `sqlj.runtime.Scrollable`.

with-clause
See SQLJ with-clause for a description of this clause.

positioned-iterator-column-declarations
Specifies a list of Java data types, which are the data types of the columns in the positioned iterator. The data types in the list must be separated by commas. The order of the data types in the positioned iterator declaration is the same as the order of the columns in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See Java, JDBC, and SQL data types for a list of compatible data types.

named-iterator-column-declarations
Specifies a list of Java data types and Java identifiers, which are the data types and names of the columns in the named iterator. Pairs of data types and names must be separated by commas. The name of a column in the iterator must match, except for case, the name of a column in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See Java, JDBC, and SQL data types for a list of compatible data types.

Usage notes:

- An iterator declaration clause can appear anywhere in a Java program that a Java class declaration can appear.
- When a named iterator declaration contains more than one pair of Java data types and Java IDs, all Java IDs within the list must be unique.

Related tasks:

- "Connecting to a data source using SQLJ" on page 322

Related reference:

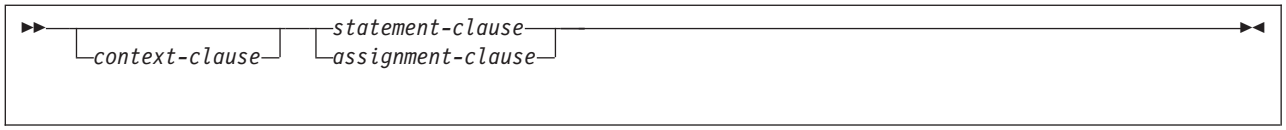
- "SQLJ implements-clause" on page 396
- "SQLJ with-clause" on page 397

SQLJ executable-clause

An executable clause contains an SQL statement or an assignment statement. An assignment statement assigns the result of an SQL operation to a Java variable.

This topic describes the general form of an executable clause.

Syntax:



Usage notes:

- An executable clause can appear anywhere in a Java program that a Java statement can appear.
- SQLJ reports negative SQL codes from executable clauses through class `java.sql.SQLException`.

  If SQLJ raises a run-time exception during the execution of an executable clause, the value of any host expression of type OUT or INOUT is undefined.

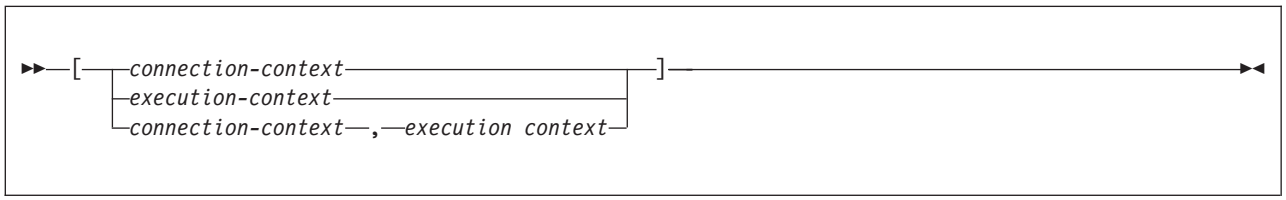
Related reference:

- "SQLJ assignment-clause" on page 405
- "SQLJ context-clause" on page 402
- "SQLJ statement-clause" on page 403

SQLJ context-clause

A context clause specifies a connection context, an execution context, or both. You use a connection context to connect to a data source. You use an execution context to monitor and modify SQL statement execution.

Syntax:



Description:

connection-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

execution-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of class `sqlj.runtime.ExecutionContext`.

Usage notes:

- If you do not specify a connection context in an executable clause, SQLJ uses the default connection context.
- If you do not specify an execution context, SQLJ obtains the execution context from the connection context of the statement.

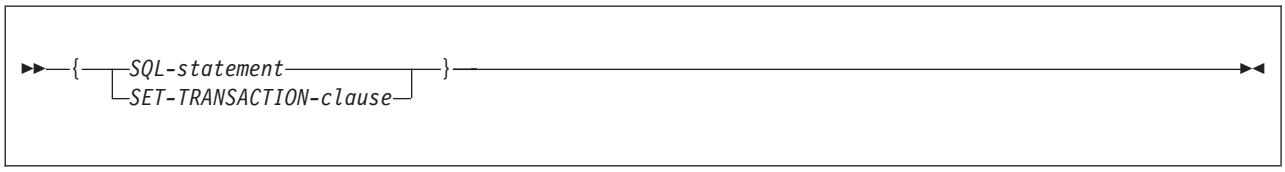
Related tasks:

- "Connecting to a data source using SQLJ" on page 322
- "Controlling the execution of SQL statements in SQLJ" on page 353

SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax:



Description:

SQL-statement
You can include the DB2 UDB for Linux, UNIX and Windows SQL statements in Table 73 in a statement clause.

SET-TRANSACTION-clause
Sets the isolation level for SQL statements in the program and the access mode for the connection. The SET TRANSACTION clause is equivalent to the SET TRANSACTION statement, which is described in the ANSI/ISO SQL standard of 1992 and is supported in some implementations of SQL. See SQLJ SET-TRANSACTION-clause for more information.

Table 73. Valid SQL statements in an SQLJ statement clause

ALTER DATABASE
ALTER FUNCTION
ALTER INDEX
ALTER PROCEDURE
ALTER STOGROUP
ALTER TABLE
ALTER TABLESPACE
CALL
COMMENT ON
COMMIT
CREATE ALIAS
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE GLOBAL TEMPORARY TABLE
CREATE INDEX
CREATE PROCEDURE
CREATE STOGROUP
CREATE SYNONYM
CREATE TABLE
CREATE TABLESPACE
CREATE TRIGGER
CREATE VIEW
DECLARE GLOBAL TEMPORARY TABLE
DELETE
DROP ALIAS
DROP DATABASE
DROP DISTINCT TYPE
DROP FUNCTION
DROP INDEX
DROP PACKAGE

Table 73. Valid SQL statements in an SQLJ statement clause (continued)

DROP PROCEDURE
DROP STOGROUP
DROP SYNONYM
DROP TABLE
DROP TABLESPACE
DROP TRIGGER
DROP VIEW
FETCH
GRANT
INSERT
LOCK TABLE
REVOKE
ROLLBACK
SAVEPOINT
SELECT INTO
SET CURRENT DEFAULT TRANSFORM GROUP
SET CURRENT DEGREE
SET CURRENT EXPLAIN MODE
SET CURRENT EXPLAIN SNAPSHOT
SET CURRENT ISOLATION
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
SET CURRENT OPTIMIZATION HINT
SET CURRENT PACKAGESET (USER is not supported)
SET CURRENT PRECISION
SET CURRENT QUERY OPTIMIZATION
SET CURRENT REFRESH AGE
SET CURRENT SCHEMA
SET PATH
UPDATE

Usage notes:

- SQLJ supports both positioned and searched DELETE and UPDATE operations.
- For a FETCH statement, a positioned DELETE statement, or a positioned UPDATE statement, you must use an iterator to refer to rows in a result table.

Related tasks:

- "Setting the isolation level for an SQLJ transaction" on page 327

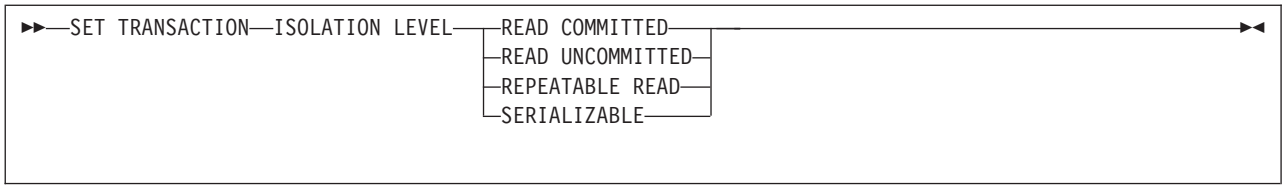
Related reference:

- "SQLJ SET-TRANSACTION-clause" on page 404

SQLJ SET-TRANSACTION-clause

The SET TRANSACTION clause sets the isolation level for the current unit of work.

Syntax:



Description:

ISOLATION LEVEL
Specifies one of the following isolation levels:

READ COMMITTED
Specifies that the current DB2 isolation level is cursor stability.

READ UNCOMMITTED
Specifies that the current DB2 isolation level is uncommitted read.

REPEATABLE READ
Specifies that the current DB2 isolation level is read stability.

SERIALIZABLE
Specifies that the current DB2 isolation level is repeatable read.

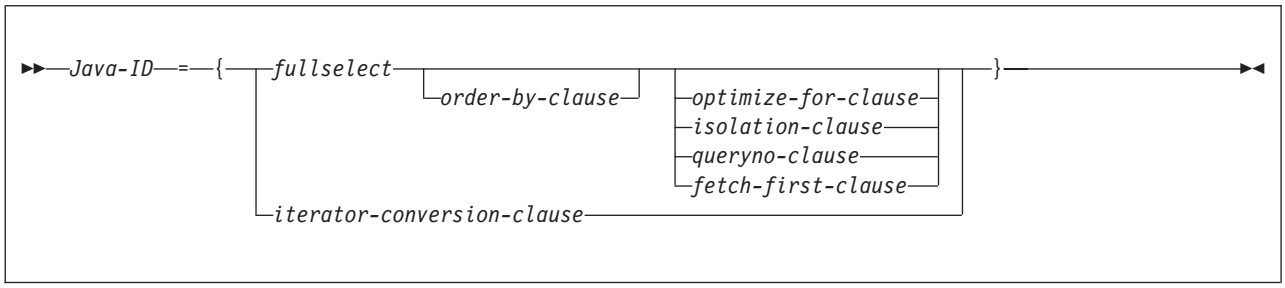
Usage notes:

You can execute SET TRANSACTION only at the beginning of a transaction.

SQLJ assignment-clause

The assignment clause assigns the result of an SQL operation to a Java variable.

Syntax:



Description:

Java-ID
Identifies an iterator that was declared previously as an instance of an iterator class.

fullselect
Generates a result table.

iterator-conversion-clause
See SQLJ iterator-conversion-clause for a description of this clause.

Usage notes:

- If the object that is identified by *Java-ID* is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must be compatible with

the data type of the corresponding column in the iterator. See Java, JDBC, and SQL data types for a list of compatible Java and SQL data types.

- If the object that is identified by *Java-ID* is a named iterator, the name of each accessor method must match, except for case, the name of a column in the result set, except for case. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the result set.
- You can put an assignment clause anywhere in a Java program that a Java assignment statement can appear. However, you cannot put an assignment clause where a Java assignment expression can appear. For example, you cannot specify an assignment clause in the control list of a for statement.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 345

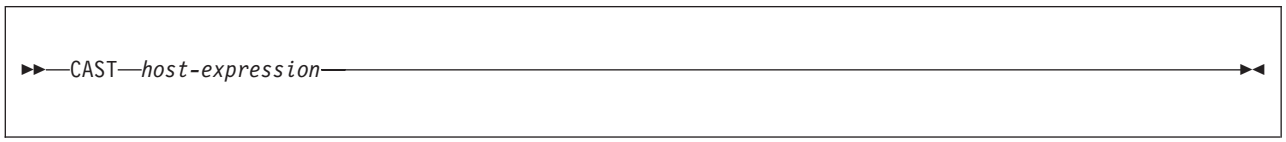
Related reference:

- "Fullselect" in the *SQL Reference, Volume 1*
- "Select-statement" in the *SQL Reference, Volume 1*
- "SQLJ iterator-conversion-clause" on page 406

SQLJ iterator-conversion-clause

The iterator conversion clause converts a JDBC `ResultSet` to an iterator.

Syntax:



Description:

host-expression
Identifies the JDBC `ResultSet` that is to be converted to an SQLJ iterator.

Usage notes:

- If the iterator to which the JDBC `ResultSet` is to be converted is a positioned iterator, the number of columns in the `ResultSet` must match the number of columns in the iterator. In addition, the data type of each column in the `ResultSet` must be compatible with the data type of the corresponding column in the iterator.
- If the iterator is a named iterator, the name of each accessor method must match, except for case, the name of a column in the `ResultSet`. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the `ResultSet`.
- When an iterator that is generated through the iterator conversion clause is closed, the `ResultSet` from which the iterator is generated is also closed.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 345

Selected sqlj.runtime classes and interfaces

The `sqlj.runtime` package defines the run-time classes and interfaces that SQLJ uses. This topic describes:

- Each class of `sqlj.runtime` that contains methods that you can invoke in your SQLJ application programs
- Each of the interfaces that you might need to implement in your SQLJ application programs

sqlj.runtime.ExecutionContext class:

The `sqlj.runtime.ExecutionContext` class is defined for execution contexts. You can use an execution context to control the execution of SQL statements. After you declare an execution context and create an instance of that execution context, you can use the following methods.

executeBatch

Format:

```
public synchronized int[] executeBatch()
```

Executes the pending statement batch and returns an array of update counts. If no pending statement batch exists, null is returned. When this method is called, the statement batch is cleared, even if the call results in an exception.

getBatchLimit

Format:

```
synchronized public int getBatchLimit()
```

Returns the current batch limit, which is the number of statements that are added to a batch before the batch is implicitly executed.

getBatchUpdateCounts

Format:

```
public synchronized int[] getBatchUpdateCounts()
```

Returns an array that contains the number of rows updated by each statement that successfully executed in a batch. Returns null if no statements in the batch completed successfully.

getMaxFieldSize

Format:

```
public int getMaxFieldSize()
```

Returns the maximum number of bytes that are returned for any character or binary column in queries that use the given execution context. A value of 0 means that the maximum number of bytes is unlimited.

getMaxRows

Format:

```
public int getMaxRows()
```

Returns the maximum number of rows that are returned for any query that uses the given execution context. A value of 0 means that the maximum number of rows is unlimited.

getNextResultSet

Formats:

```
public ResultSet getNextResultSet()
public ResultSet getNextResultSet(int current)
```

After a stored procedure call, returns a result set from the stored procedure. A value of null means that there are no more result sets to be returned.

When you invoke `getNextResultSet()`, SQLJ closes the currently-open result set and advances to the next result set. When you invoke `getNextResultSet(int current)`, the value of *current* indicates what SQLJ does with the currently-open result set before it advances to the next result set:

java.sql.Statement.CLOSE_CURRENT_RESULT
Specifies that the current `ResultSet` object is closed when the next `ResultSet` object is returned.

java.sql.Statement.KEEP_CURRENT_RESULT
Specifies that the current `ResultSet` object stays open when the next `ResultSet` object is returned.

java.sql.Statement.CLOSE_ALL_RESULTS
Specifies that all open `ResultSet` objects are closed when the next `ResultSet` object is returned.

`getNextResultSet(int current)` requires JDK 1.4 or later.

getUpdateCount
Format:

```
public abstract int getUpdateCount() throws SQLException
```

Returns:

ExecutionContext.ADD_BATCH_COUNT
If the statement was added to an existing batch.

ExecutionContext.NEW_BATCH_COUNT
If the statement was the first statement in a new batch.

ExecutionContext.EXEC_BATCH_COUNT
If the statement was part of a batch, and the batch was executed.

Other integer
If the statement was executed rather than added to a batch. This value is the number of rows that were updated by the statement.

getWarnings
Format:

```
public SQLWarning getWarnings()
```

Returns the first warning that was reported by the last SQL operation that was executed using this context. Subsequent warnings are chained to the first warning.

Use this method to retrieve positive SQLCODEs.

isBatching
Format:

```
public synchronized boolean isBatching()
```

Returns `true` if batching is enabled. Returns `false` if batching is disabled.

setBatching

Format:

```
public synchronized void setBatching(boolean)
```

Enables or disables batching.

setBatchLimit

Format:

```
public synchronized void setBatchLimit(int)
```

Sets the maximum number of statements that are added to a batch before the batch is implicitly executed. Possible values for the input parameter are:

ExecutionContext.UNLIMITED_BATCH
Indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking `setBatchLimit`.

ExecutionContext.AUTO_BATCH
Indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer
The number of statements that are added to the batch before SQLJ executes the batch implicitly. The batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

setMaxFieldSize

Format:

```
public void setMaxFieldSize(int max)
```

Specifies the maximum number of bytes that are returned for any character or binary column in queries that use the given execution context. The default is 0, which means that the maximum number of bytes is unlimited.

setMaxRows

Format:

```
public void setMaxRows(int max)
```

Specifies the maximum number of rows that are returned for any query that uses the given execution context. The default is 0, which means that the maximum number of rows returned is unlimited.

sqlj.runtime.ConnectionContext interface:

`sqlj.runtime.ConnectionContext` is an interface that SQLJ implements when you execute a connection declaration clause and thereby create a connection context class.

Suppose that you declare a connection named Ctx. You can then use the following methods to determine or change the default context.

getDefaultContext

Format:

```
public static Ctx getDefaultContext()
```

Returns the default connection context object for the Ctx class.

SetDefaultContext

Format:

```
public static void Ctx setDefaultContext(Ctx default-context)
```

Sets the default connection context object for the Ctx class.

sqlj.runtime.ForUpdate interface:

Implement the `sqlj.runtime.ForUpdate` interface for positioned UPDATE or DELETE operations. You implement `sqlj.runtime.ForUpdate` in an SQLJ iterator declaration clause.

sqlj.runtime.NamedIterator interface:

`sqlj.runtime.NamedIterator` is an interface that SQLJ implements when you declare a named iterator. When you declare an instance of a named iterator, SQLJ creates an accessor method for each column in the expected result table. An accessor method returns the data from its column of the result table. The name of an accessor method matches the name of the corresponding column in the named iterator.

In addition to the accessor methods, SQLJ generates the following methods that you can invoke in your SQLJ application.

close

Format:

```
public abstract void close() throws SQLException
```

Releases database resources that the iterator uses.

isClosed

Format:

```
public abstract boolean isClosed() throws SQLException
```

Returns a value of `true` if the `close` method has been invoked.

next

Format:

```
public abstract boolean next() throws SQLException
```

Advances the iterator to the next row. Before an instance of the `next` method is invoked for the first time, the iterator is positioned before the first row of the result table. `next` returns a value of `true` when a next row is available and `false` when all rows have been retrieved.

sqlj.runtime.PositionedIterator interface:

`sqlj.runtime.PositionedIterator` is an interface that SQLJ implements when you declare a positioned iterator. After you declare and create an instance of a positioned iterator, you can use the following method.

endFetch

Format:

```
public abstract boolean endFetch() throws SQLException
```

Returns a value of `true` if the iterator is not positioned on a row.

sqlj.runtime.ResultSetIterator interface:

`sqlj.runtime.ResultSetIterator` is an interface that SQLJ implements when you declare an iterator. After you declare and create an instance of an iterator, you can use the following methods.

clearWarnings
Format:

```
public abstract void clearWarnings() throws SQLException
```

Returns null until a new warning is reported for this iterator.

close
Format:

```
public abstract void close() throws SQLException
```

Releases database resources that the iterator uses.

getResultSet
Format:

```
public abstract ResultSet getResultSet() throws SQLException
```

Returns a JDBC result set representation of an SQLJ iterator.

getWarnings
Format:

```
public abstract SQLWarning getWarnings() throws SQLException
```

Returns the first warning that is reported by calls on this iterator. Subsequent iterator warnings are be chained to this SQLWarning. The warning chain is automatically cleared each time a new row is read.

isClosed
Format:

```
public abstract boolean isClosed() throws SQLException
```

Returns a value of `true` if the `close` method has been invoked.

next
Format:

```
public abstract boolean next() throws SQLException
```

Advances the iterator to the next row. Before an instance of the `next` method is invoked for the first time, the iterator is positioned before the first row of the result table. `next` returns a value of `true` when a next row is available and `false` when all rows have been retrieved.

sqlj.runtime.Scrollable interface:

`sqlj.runtime.Scrollable` is an interface that you implement when you declare a scrollable iterator. You use the `sqlj.runtime.Scrollable` methods to move around in the result table and to check your position in the result table.

absolute(int)
Format:

```
public abstract boolean absolute (int n) throws SQLException
```

Moves the iterator to a specified row.

If *n*>0, positions the iterator on row *n* of the result table. If *n*<0, and *m* is the number of rows in the result table, positions the iterator on row *m+n+1* of the result table.

If the absolute value of *n* is greater than the number of rows in the result table, positions the cursor after the last row if *n* is negative, or before the first row if *n* is positive.

`Absolute(1)` is the same as `first()`. `Absolute(-1)` is the same as `last()`.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

afterLast()

Format:

```
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

beforeFirst()

Format:

```
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

first()

Format:

```
public abstract boolean first() throws SQLException
```

Moves the iterator to the first row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

getFetchDirection()

Format:

```
public abstract int getFetchDirection ( ) throws SQLException
```

Returns the fetch direction of the iterator. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of processing is not known.

isAfterLast()

Format:

```
public abstract boolean isAfterLast() throws SQLException
```

Returns `true` if the iterator is positioned after the last row of the result table. Otherwise, returns `false`.

isBeforeFirst()

Format:

```
public abstract boolean isBeforeFirst() throws SQLException
```

Returns `true` if the iterator is positioned before the first row of the result table. Otherwise, returns `false`.

isFirst()

Format:

```
public abstract boolean isFirst() throws SQLException
```

Returns `true` if the iterator is positioned on the first row of the result table. Otherwise, returns `false`.

isLast()

Format:

```
public abstract boolean isLast() throws SQLException
```

Returns `true` if the iterator is positioned on the last row of the result table. Otherwise, returns `false`.

last()

Format:

```
public abstract boolean last() throws SQLException
```

Moves the iterator to the last row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

previous()

Format:

```
public abstract boolean previous() throws SQLException
```

Moves the iterator to the previous row of the result table.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

relative(int)

Format:

```
public abstract boolean relative(int n) throws SQLException
```

If $n>0$, positions the iterator on the row that is n rows after the current row. If $n<0$, positions the iterator on the row that is n rows before the current row. If $n=0$, positions the iterator on the current row.

The cursor must be on a valid row of the result table before you can use this method. If the cursor is before the first row or after the last throw, the method throws an `SQLException`.

Suppose that m is the number of rows in the result table and x is the current row number in the result table. If $n>0$ and $x+n>m$, the the iterator is positioned after the last row. If $n<0$ and $x+n<1$, the iterator is positioned before the first row.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

setFetchDirection(int)

Format:

```
public abstract void setFetchDirection (int) throws SQLException
```

Gives the SQLJ runtime environment a hint as to the direction in which rows of this iterator object are processed. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of processing is not known.

Related tasks:

- "Making batch updates in SQLJ applications" on page 355
- "Connecting to a data source using SQLJ" on page 322
- "Using a named iterator in an SQLJ application" on page 332
- "Using a positioned iterator in an SQLJ application" on page 334
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 336
- "Using scrollable iterators in an SQLJ application" on page 361
- "Handling SQL warnings in an SQLJ application" on page 344
- "Controlling the execution of SQL statements in SQLJ" on page 353

DB2 Universal JDBC Driver reference information

The sections that follow contain information that is specific to the DB2 Universal JDBC Driver.

Summary of DB2 Universal JDBC Driver extensions to JDBC

This topic describes the JDBC APIs that are specific to the DB2 Universal JDBC Driver.

To use any of the methods that are described in this topic, you must cast an instance of the related, standard JDBC class to an instance of the DB2-only class. For example:

```
javax.sql.DataSource ds =
 new com.ibm.db2.jcc.DB2SimpleDataSource();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

DB2ActiveServerList class:

The `com.ibm.db2.jcc.DB2ActiveServerList` class implements the `java.io.Serializable` and `javax.naming.Referenceable` interfaces.

DB2ActiveServerList methods:

getAlternatePortNumber
Format:

```
public int[] getAlternatePortNumber()
```

Retrieves the port numbers that are associated with the alternate DB2 UDB servers.

getAlternateServerName
Format:

```
public String[] getAlternateServerName()
```

Retrieves an array that contains the names of the alternate DB2 UDB servers. These values are IP addresses or DNS server names.

setAlternatePortNumber

Format:

```
public void setAlternatePortNumber(int[] alternatePortNumberList)
```

Sets the port numbers that are associated with the alternate DB2 UDB servers.

setAlternateServerName

Format:

```
public void setAlternateServerName(String[] alternateServer)
```

Sets the alternate server names for DB2 UDB servers. These values are IP addresses or DNS server names.

DB2BaseDataSource class:

The `com.ibm.db2.jcc.DB2BaseDataSource` class is the abstract data source parent class for all DB2-specific implementations of `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource`.

DB2BaseDataSource properties:

The following properties are defined only for the DB2 Universal JDBC Driver. See Properties for the DB2 Universal JDBC Driver for explanations of these properties.

Each of these properties has a `setXXX` method to set the value of the property and a `getXXX` method to retrieve the value. A `setXXX` method has this form:

```
void setProperty-name(data-type property-value)
```

A `getXXX` method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name, with the first character capitalized.

Table 74 lists the DB2 Universal JDBC Driver properties and their data types.

Table 74. DB2 Universal JDBC Driver properties and their data types

Property name	Data type
com.ibm.db2.jcc.DB2BaseDataSource.activeServerListJNDIName	String
com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation	String
com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation	String
com.ibm.db2.jcc.DB2BaseDataSource.clientUser	String
com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation	String
com.ibm.db2.jcc.DB2BaseDataSource.cliSchema	String
com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	String
com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout	int
com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	String
com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity	int
com.ibm.db2.jcc.DB2BaseDataSource.currentSchema	String
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	String

Table 74. DB2 Universal JDBC Driver properties and their data types (continued)

Property name	Data type
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	String
com.ibm.db2.jcc.DB2BaseDataSource.databaseName	String
com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares	boolean
com.ibm.db2.jcc.DB2BaseDataSource.description	String
com.ibm.db2.jcc.DB2BaseDataSource.driverType	int
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData	boolean
com.ibm.db2.jcc.DB2BaseDataSource.gssCredential	Object
com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection	String
com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic	int
com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal	String
com.ibm.db2.jcc.DB2BaseDataSource.logWriter	PrintWriter
com.ibm.db2.jcc.DB2BaseDataSource.portNumber	int
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability	int
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	int
com.ibm.db2.jcc.DB2BaseDataSource.serverName	String
com.ibm.db2.jcc.DB2BaseDataSource.readOnly	boolean
com.ibm.db2.jcc.DB2BaseDataSource.traceFile	String
com.ibm.db2.jcc.DB2BaseDataSource.traceLevel	int
com.ibm.db2.jcc.DB2BaseDataSource.user	String

DB2BaseDataSource methods:

In addition to the `getXXX` and `setXXX` methods for the DB2BaseDataSource properties, the following methods are defined only for the DB2 Universal JDBC Driver.

getReference

Format:

```
public javax.naming.Reference getReference()
  throws javax.naming.NamingException
```

Retrieves the `Reference` of a `DataSource` object. For an explanation of a `Reference`, see the description of `javax.naming.Referenceable` in the JNDI documentation at:

`http://java.sun.com/products/jndi/docs.html`

DB2Connection interface:

The `com.ibm.db2.jcc.DB2Connection` interface extends the `java.sql.Connection` interface.

DB2Connection methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

getDB2ClientAccountingInformation

Format:

```
public String getDB2ClientAccountingInformation()
  throws SQLException
```

Returns accounting information for the current client.

getDB2ClientApplicationInformation

Format:

```
public String getDB2ClientApplicationInformation()
  throws SQLException
```

Returns application information for the current client.

getDB2ClientUser

Format:

```
public String getDB2ClientUser()
  throws SQLException
```

Returns the current client user name for the connection. This name is not the user value for the JDBC connection.

getDB2ClientWorkstation

Format:

```
public String getDB2ClientWorkstation()
  throws SQLException
```

Returns current client workstation name for the current client.

getDB2CurrentPackagePath

Format:

```
public String getDB2CurrentPackagePath()
  throws SQLException
```

Returns the list of DB2 package collections that are searched for the DB2 Universal JDBC Driver packages.

getDB2CurrentPackageSet

Format:

```
public String getDB2CurrentPackageSet()
  throws SQLException
```

Returns the collection ID for the connection.

getDB2SystemMonitor

Format:

```
public abstract DB2SystemMonitor getDB2SystemMonitor()
  throws SQLException
```

Returns the system monitor object for the connection. Each DB2 Universal JDBC Driver connection can have a single system monitor. See "DB2SystemMonitor interface" on page 423 for more information.

getJccLogWriter

Format:

```
public PrintWriter getJccLogWriter()
  throws SQLException
```

Returns the current trace destination for the DB2 Universal JDBC Driver trace.

setDB2ClientAccountingInformation

Format:

```
public void setDB2ClientAccountingInformation(String info)
  throws SQLException
```

Specifies accounting information for the connection. This information is for client accounting purposes. This value can change during a connection.

Parameter description:

info
User-specified accounting information. The maximum length depends on the server. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2ClientApplicationInformation

Format:

```
public void setDB2ClientApplicationInformation(String info)
  throws SQLException
```

Specifies application information for the connection. This information is for client accounting purposes. This value can change during a connection.

Parameter description:

info
User-specified application information. The maximum length depends on the server. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2ClientUser

Format:

```
public void setDB2ClientUser(String user)
  throws SQLException
```

Specifies the current client user name for the connection. This name is for client accounting purposes, and is not the user value for the JDBC connection. Unlike the user for the JDBC connection, the current client user name can change during a connection.

Parameter description:

user
The user ID for the current client.The maximum length depends on the server. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2ClientWorkstation

Format:

```
public void setDB2ClientWorkstation(String name)
  throws SQLException
```

Specifies the current client workstation name for the connection. This name is for client accounting purposes. The current client workstation name can change during a connection.

Parameter description:

name
The workstation name for the current client.The maximum length depends on the server. For a DB2 UDB for Linux, UNIX and Windows server, the maximum length is 255 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2CurrentPackagePath
Format:

```
public void setDB2CurrentPackagePath(String packagePath)
  throws SQLException
```

Specifies a list of collection IDs that DB2 searches for the DB2 Universal JDBC Driver DB2 packages.

Parameter description:

packagePath
A comma-separated list of collection IDs.

setDB2CurrentPackageSet
Format:

```
public void setDB2CurrentPackageSet(String packageSet)
  throws SQLException
```

Specifies the collection ID for the connection. When you set this value, you also set the collection ID of the DB2 Universal JDBC Driver instance that is used for the connection.

Parameter description:

packageSet
The collection ID for the connection. The maximum length for the packageSet value is 18 bytes. You can invoke this method as an alternative to executing the SQL SET CURRENT PACKAGESET statement in your program.

setJccLogWriter
Formats:

```
public void setJccLogWriter(PrintWriter logWriter)
  throws SQLException

public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
  throws SQLException
```

Enables or disables the DB2 Universal JDBC Driver trace, or changes the trace destination during an active connection.

Parameter descriptions:

logWriter
An object of type `java.io.PrintWriter` to which the DB2 Universal JDBC Driver writes trace output. To turn off the trace, set the value of *logWriter* to `null`.

traceLevel
Specifies the types of traces to collect. See the description of the traceLevel property in Properties for the DB2 Universal JDBC Driver for valid values.

DB2DatabaseMetaData interface:

The com.ibm.db2.jcc.DB2DatabaseMetaData interface extends the java.sql.DatabaseMetaData interface.

DB2DatabaseMetaData methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

DB2Diagnosable interface:

The com.ibm.db2.jcc.DB2Diagnosable interface provides a mechanism for getting DB2 diagnostics from a DB2 SQLException.

DB2Diagnosable methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

getSqlca
Format:

```
public DB2Sqlca getSqlca()
```

Returns a DB2Sqlca object from a java.sql.Exception that is produced under a DB2 Universal JDBC Driver.

getThrowable
Format:

```
public Throwable getThrowable()
```

Returns a java.lang.Throwable object from a java.sql.Exception that is produced under a DB2 Universal JDBC Driver.

printTrace
Format:

```
static public void printTrace(java.io.PrintWriter printWriter,
  String header)
```

Prints diagnostic information after a java.sql.Exception is thrown under a DB2 Universal JDBC Driver.

Parameter descriptions:

printWriter
The destination for the diagnostic information.

header
User-defined information that is printed at the beginning of the output.

DB2ExceptionFormatter class:

The com.ibm.db2.jcc.DB2ExceptionFormatter class contains methods for printing diagnostic information to a stream.

DB2ExceptionFormatter methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

printTrace
Formats:

```
static public void printTrace(java.sql.SQLException sqlException,
  java.io.PrintWriter printWriter, String header)

static public void printTrace(DB2Sqlca sqlca,
  java.io.PrintWriter printWriter, String header)

static public void printTrace(java.lang.Throwable throwable,
  java.io.PrintWriter printWriter, String header)
```

Prints diagnostic information after an exception is thrown.

Parameter descriptions:

sqlException | sqlca | throwable
The exception that was thrown during a previous JDBC or Java operation.

printWriter
The destination for the diagnostic information.

header
User-defined information that is printed at the beginning of the output.

DB2RowID interface:

The `com.ibm.db2.jcc.DB2RowID` class is used for declaring Java objects for use with the DB2 ROWID data type.

DB2RowID methods:

The following method is defined only for the DB2 Universal JDBC Driver.

getBytes
Format:

```
public byte[] getBytes()
```

Converts a com.ibm.jcc.DB2RowID object to bytes.

DB2SimpleDataSource class:

The `com.ibm.db2.jcc.DB2SimpleDataSource` class extends the `DataBaseDataSource` class. A `DataBaseDataSource` object does not support connection pooling or distributed transactions. It contains all of the properties and methods that the `DB2BaseDataSource` class contains. In addition, `DB2SimpleDataSource` contains the following DB2 Universal JDBC Driver-only properties.

DB2SimpleDataSource properties:

The following property is defined only for the DB2 Universal JDBC Driver. See Properties for the DB2 Universal JDBC Driver for an explanation of this property.
String com.ibm.db2.jcc.DB2SimpleDataSource.password

DB2SimpleDataSource methods:

The following method is defined only for the DB2 Universal JDBC Driver.

setPassword
Format:

```
public void setPassword(String password)
```

Sets the password for the DB2SimpleDataSource object. There is no corresponding getPassword method. Therefore, the password cannot be encrypted because there is no way to retrieve the password so that you can decrypt it.

DB2Sqlca class:

The com.ibm.db2.jcc.DB2Sqlca class is an encapsulation of the DB2 SQLCA. .

DB2Sqlca methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

getMessage
Format:

```
public abstract String getMessage()
```

Returns error message text.

getSqlCode
Format:

```
public abstract int getSqlCode()
```

Returns an SQL error code value.

getSqlErrd
Format:

```
public abstract int[] getSqlErrd()
```

Returns an array, each element of which contains an SQLCA SQLERRD.

getSqlErrmc
Format:

```
public abstract String getSqlErrmc()
```

Returns a string that contains the SQLCA SQLERRMC values, delimited with spaces.

getSqlErrmcTokens
Format:

```
public abstract String[] getSqlErrmcTokens()
```

Returns an array, each element of which contains an SQLCA SQLERRMC token.

getSqlErrd
Format:

```
public abstract int[] getSqlErrd()
```

Returns an array, each element of which contains an SQLCA SQLERRP value.

getSqlErrp
Format:

```
public abstract String getSqlErrp()
```

Returns the SQLCA SQLERRP value.

getSqlState
Format:

```
public abstract String getSqlState()
```

Returns the SQLCA SQLSTATE value.

getSqlWarn

Format:

```
public abstract char[] getSqlWarn()
```

Returns an array, each element of which contains an SQLCA SQLWARN value.

DB2SystemMonitor interface:

The `com.ibm.db2.jcc.DB2SystemMonitor` interface is used for collecting system monitoring data for a connection. Each connection can have one `DB2SystemMonitor` instance.

DB2SystemMonitor fields:

The following fields are defined only for the DB2 Universal JDBC Driver.

public final static int RESET_TIMES
public final static int ACCUMULATE_TIMES

These values are arguments for the `DB2SystemMonitor.start` method. RESET_TIMES sets time counters to zero before monitoring starts. ACCUMULATE_TIMES does not set time counters to zero.

DB2SystemMonitor methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

enable

Format:

```
public void enable(boolean on)
  throws java.sql.SQLException
```

Enables the system monitor that is associated with a connection. This method cannot be called during monitoring. All times are reset when `enable` is invoked.

getApplicationTimeMillis

Format:

```
public long getApplicationTimeMillis()
  throws java.sql.SQLException
```

Returns the sum of the application, JDBC driver, network I/O, and DB2 server elapsed times. The time is in milliseconds.

A monitored elapsed time interval is the difference, in milliseconds, between these points in the JDBC driver processing:

Interval beginning
When `start` is called.

Interval end
When `stop` is called.

`getApplicationTimeMillis` returns 0 if system monitoring is disabled. Calling this method without first calling the `stop` method results in an `SQLException`.

getCoreDriverTimeMicros

Format:

```
public long getCoreDriverTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of elapsed monitored API times that were collected while system monitoring was enabled. The time is in microseconds.

A monitored API is a JDBC driver method for which processing time is collected. In general, elapsed times are monitored only for APIs that might result in network I/O or DB2 server interaction. For example, `PreparedStatement.setXXX` methods and `ResultSet.getXXX` methods are not monitored.

Monitored API elapsed time includes the total time that is spent in the driver for a method call. This time includes any network I/O time and DB2 server elapsed time.

A monitored API elapsed time interval is the difference, in microseconds, between these points in the JDBC driver processing:

Interval beginning
When a monitored API is called by the application.

Interval end
Immediately before the monitored API returns control to the application.

`getCoreDriverTimeMicros` returns 0 if system monitoring is disabled. Calling this method without first calling the `stop` method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an `SQLException`.

getNetworkIOTimeMicros

Format:

```
public long getNetworkIOTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of elapsed network I/O times that were collected while system monitoring was enabled. The time is in microseconds.

Elapsed network I/O time includes the time to write and read DRDA data from network I/O streams. A network I/O elapsed time interval is the time interval to perform the following operations in the JDBC driver:

- Issue a TCP/IP command to send a DRDA message to the DB2 server. This time interval is the difference, in microseconds, between points immediately before and after a write and flush to the network I/O stream is performed.
- Issue a TCP/IP command to receive DRDA reply messages from the DB2 server. This is time interval is the difference, in microseconds, between points immediately before and after a read on the network I/O stream is performed.

Network I/O time intervals are captured for all send and receive operations, including the sending of messages for commits and rollbacks.

The time spent waiting for network I/O might be impacted by delays in CPU dispatching at the DB2 server for low-priority SQL requests. Network I/O time intervals include DB2 server elapsed time.

getNetworkIOTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an SQLException.

getServerTimeMicros

Format:

```
public long getServerTimeMicros()
  throws java.sql.SQLException
```

Returns the sum of all reported DB2 server elapsed times that were collected while system monitoring was enabled. The time is in microseconds.

The DB2 server reports elapsed times under these conditions:

- The server supports returning elapsed time data to the client.
- The server performs operations that can be monitored. For example, DB2 server elapsed time is not returned for commits or rollbacks.

DB2 server elapsed time is defined as the elapsed time to parse the request data stream, process the command, and generate the reply data stream at the server. Network time to receive or send the data stream is not included.

a DB2 server elapsed time interval is the difference, in microseconds, between these points in the server processing:

Interval beginning
When the operating system dispatches DB2 to process a TCP/IP message that is received from the JDBC driver.

Interval end
When DB2 is ready to issue the TCP/IP command to return the reply message to the client.

getServerTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in an SQLException.

start

Format:

```
public void start (int lapMode)
  throws java.sql.SQLException
```

If the system monitor is enabled, start begins the collection of system monitoring data for a connection. Valid values for *lapMode* are RESET_TIMES or ACCUMULATE_TIMES.

Calling this method with system monitoring disabled does nothing. Calling this method more than once without an intervening stop call results in an SQLException.

stop

Format:

```
public void stop()
  throws java.sql.SQLException
```

If the system monitor is enabled, stop ends the collection of system monitoring data for a connection. After monitoring is stopped, monitored times can be obtained with the getXXX methods of DB2SystemMonitor.

Calling this method with system monitoring disabled does nothing. Calling this method without first calling `start`, or calling this method more than once without an intervening `start` call results in an `SQLException`.

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270
- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282

Related reference:

- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*
- "Properties for the DB2 Universal JDBC Driver" on page 370

JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers

The DB2 Universal JDBC Driver differs from the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) in the following areas:

Supported methods:

The DB2 Universal JDBC Driver supports a number of JDBC methods that the other drivers do not support, and does not support several methods that the other drivers support. For details, see Comparison of driver support for JDBC APIs.

Support for scrollable and updatable ResultSets:

The DB2 Universal JDBC Driver supports scrollable and updatable `ResultSets`.

The DB2 JDBC Type 2 Driver supports scrollable `ResultSets` but not updatable `ResultSets`.

Difference in URL syntax:

The syntax of the *url* parameter in the `DriverManager.getConnection` method is different for each driver. See the following topics for more information:

- Connect to a data source using the DriverManager interface with the DB2 Universal JDBC Driver
- Connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver

Difference in error codes and SQLSTATEs returned for driver errors:

The DB2 Universal JDBC Driver does not use existing SQLCODEs or SQLSTATEs for internal errors, as the other drivers do. See Error codes issued by the DB2 Universal JDBC Driver and SQLSTATEs issued by the DB2 Universal JDBC Driver.

The JDBC/SQLJ driver for z/OS return ODBC SQLSTATEs when internal errors occur.

How much error message text is returned:

With the DB2 Universal JDBC Driver, when you execute `SQLException.getMessage()`, formatted message text is not returned unless you set the `retrieveMessagesFromServerOnGetMessage` property to `true`.

With the DB2 JDBC Type 2 Driver, when you execute `SQLException.getMessage()`, formatted message text is returned.

Security mechanisms:

The JDBC drivers have different security mechanisms.

For information on DB2 Universal JDBC Driver security mechanisms, seeSecurity under the DB2 Universal JDBC Driver.

For information on security mechanisms for the DB2 JDBC Type 2 Driver, see Security under the DB2 JDBC Type 2 Driver.

Support for read-only connections:

With the DB2 Universal JDBC Driver, you can make a connection read-only through the `readOnly` property for a `Connection` or `DataSource` object.

The DB2 JDBC Type 2 Driver uses the `Connection.setReadOnly` value when it determines whether to make a connection read-only. However, setting `Connection.setReadOnly(true)` does not guarantee that the connection is read-only.

Results returned from ResultSet.getString for a BIT DATA column:

The DB2 Universal JDBC Driver returns data from a `ResultSet.getString` call for a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column as a lowercase hexadecimal string.

The DB2 JDBC Type 2 Driver returns the data as an uppercase hexadecimal string.

Result of an executeUpdate call that affects no rows:

The DB2 Universal JDBC Driver generates an `SQLWarning` when an `executeUpdate` call affects no rows.

The DB2 JDBC Type 2 Driver does not generate an returns an `SQLWarning`.

Result of a getDate or getTime call for a TIMESTAMP column:

The DB2 Universal JDBC Driver does not generate an `SQLWarning` when a `getDate` or `getTime` call is made against a TIMESTAMP column.

The DB2 JDBC Type 2 Driver generates an `SQLWarning` when a `getDate` or `getTime` call is made against a TIMESTAMP column.

When an exception is thrown for PreparedStatement.setXXXStream with a length mismatch:

When you use the `PreparedStatement.setBinaryStream` , `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method, the *length* parameter value must match the number of bytes in the input stream.

If the numbers of bytes do not match, the DB2 Universal JDBC Driver does not throw an exception until the subsequent `PreparedStatement.executeUpdate` method executes. Therefore, for the DB2 Universal JDBC Driver, some data might be sent to the server when the lengths to not match. That data is truncated or padded by the server. The calling application needs to issue a rollback request to undo the database updates that include the truncated or padded data.

The DB2 JDBC Type 2 Driver throws an exception after the `PreparedStatement.setBinaryStream`, `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method executes.

Default mappings for PreparedStatement.setXXXStream:

With the DB2 Universal JDBC Driver, when you use the `PreparedStatement.setBinaryStream` , `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method, and no information about the data type of the target column is available, the input data is mapped to a BLOB or CLOB data type.

For the DB2 JDBC Type 2 Driver, the input data is mapped to a VARCHAR FOR BIT DATA or VARCHAR data type.

How character conversion is done:

When character data is transferred between a client and a server, the data must be converted to a form that the receiver can process.

For the DB2 Universal JDBC Driver, character data that is sent from the database server to the client is converted using Java's built-in character converters. The conversions that the DB2 Universal JDBC Driver supports are limited to those that are supported by the underlying JRE implementation.

A DB2 Universal JDBC Driver client sends data to the database server as Unicode.

For the DB2 JDBC Type 2 Driver, character conversions can be performed if the conversions are supported by the DB2 server.

Implicit or explicit data type conversion for input parameters:

If you execute a `PreparedStatement.setXXX` method, and the resulting data type from the `setXXX` method does not match the data type of the table column to which the parameter value is assigned, the driver returns an error unless data type conversion occurs.

With the DB2 Universal JDBC Driver, conversion to the correct SQL data type occurs implicitly if the target data type is known and if the `deferPrepares` connection property is set to `false`. In this case, the implicit values override any explicit values in the `setXXX` call. If the `deferPrepares` connection property is set to `true`, you must use the `PreparedStatement.setObject` method to convert the parameter to the correct SQL data type.

For the DB2 JDBC Type 2 Driver, if the data type of a parameter does not match its default SQL data type, you must use the `PreparedStatement.setObject` method to convert the parameter to the correct SQL data type.

Support for String to BINARY conversions for input parameters:

The DB2 Universal JDBC Driver does not support `PreparedStatement.setObject` calls of the following form when *x* is an object of type `String`:

```
setObject(parameterIndex, x, java.sqlTypes.BINARY)
```

The DB2 JDBC Type 2 Driver supports calls of this type. The driver interprets the value of *x* as a hexadecimal string.

Result of PreparedStatement.setObject with a decimal scale mismatch:

With the DB2 Universal JDBC Driver, if you call `PreparedStatement.setObject` with a decimal input parameter, and the scale of the input parameter is greater than the scale of the target column, the driver truncates the trailing digits of the input value before assigning the value to the column.

The DB2 JDBC Type 2 Driver rounds the trailing digits of the input value before assigning the value to the column.

Support for conversions from the java.lang.Character data type for input parameters:

For the following form of `PreparedStatement.setObject`, the DB2 Universal JDBC Driver supports the standard data type mappings of Java objects to JDBC data types when it converts *x* to a JDBC data type:

```
setObject(parameterIndex, x)
```

The DB2 JDBC Type 2 Driver supports the non-standard mapping of *x* from `java.lang.Character` to CHAR.

Support for ResultSet.getBinaryStream against a character column:

The DB2 Universal JDBC Driver supports `ResultSet.getBinaryStream` with an argument that represents a character column only if the column has the FOR BIT DATA attribute.

For the DB2 JDBC Type 2 Driver, if the `ResultSet.getBinaryStream` argument is a character column, that column does not need to have the FOR BIT DATA attribute.

Data returned from ResultSet.getBinaryStream against a binary column:

With the DB2 Universal JDBC Driver, when you execute `ResultSet.getBinaryStream` against a binary column, the returned data is in the form of lowercase, hexadecimal digit pairs.

With the DB2 JDBC Type 2 Driver, when you execute `ResultSet.getBinaryStream` against a binary column, the returned data is in the form of uppercase, hexadecimal digit pairs.

Result of using setObject with a Boolean input type and a CHAR target type:

With the DB2 Universal JDBC Driver, when you execute `PreparedStatement.setObject(parameterIndex,x,CHAR)`, and *x* is Boolean, the value ″0″ or ″1″ is inserted into the table column.

With the DB2 JDBC Type 2 Driver, the string ″false″ or ″true″ is inserted into the table column. The table column length must be at least 5.

Result of using getBoolean to retrieve a value from a CHAR column:

With the DB2 Universal JDBC Driver, when you execute `ResultSet.getBoolean` or `CallableStatement.getBoolean` to retrieve a Boolean value from a CHAR column, and the column contains the value "false" or "0", the value `false` is returned. If the column contains any other value, `true` is returned.

With the DB2 JDBC Type 2 Driver, when you execute `ResultSet.getBoolean` or `CallableStatement.getBoolean` to retrieve a Boolean value from a CHAR column, and the column contains the value "true" or "1", the value `true` is returned. If the column contains any other value, `false` is returned.

Result of executing ResultSet.next() on a closed cursor:

With the DB2 Universal JDBC Driver, when you execute `ResultSet.next()` on a closed cursor, an `SQLException` is thrown. This conforms with the JDBC standard.

With the DB2 JDBC Type 2 Driver, when you execute `ResultSet.next()` on a closed cursor, a value of `false` is returned, and now exception is thrown.

Result of specifying null arguments in DatabaseMetaData calls:

With the DB2 Universal JDBC Driver, you can specify `null` for an argument in a `DatabaseMetaData` method call only where the JDBC specification states that `null` is allowed. Otherwise, an exception is thrown.

With the DB2 JDBC Type 2 Driver, `null` means that the argument is not used to narrow the search.

Support for DATALINKs:

The DB2 Universal JDBC Driver does not support the DATALINK SQL type.

The DB2 JDBC Type 2 Driver supports the DATALINK type in method calls of these forms:

- `PreparedStatement.setObject(`*`parameterIndex`*`, `*`x`*`, DB2Constants.DATALINK)`
- `PreparedStatement.setObject(`*`parameterIndex`*`, `*`x`*`, java.sql.Types.DATALINK)` (Java 1.4 or later)
- `PreparedStatement.setURL(` *`parameterIndex`*`, java.net.URL)`
- `PreparedStatement.setObject(` *`parameterIndex`*`, java.net.URL)`
- `PreparedStatement.setObject(`*`parameterIndex`*`, java.net.URL, java.sql.Types.DATALINK)` (Java 1.4 or later)
- `ResultSet.getString` for a DATALINK column
- `ResultSet.getURL` for a DATALINK column

Folding of method arguments to uppercase:

The DB2 Universal JDBC Driver does not fold any arguments in method calls to uppercase.

The DB2 JDBC Type 2 Driver folds the argument of a `Statement.setCursorName` call to uppercase. To prevent the cursor name from being folded to uppercase, precede and follow the cursor name with the characters \". For example:

```
Statement.setCursorName("\"mycursor\"");
```

Support for timestamp escape clauses:

The DB2 Universal JDBC Driver supports the standard form of an escape clause for `TIME`:

```
{t 'hh:mm:ss'}
```

In addition to the standard form, the DB2 JDBC Type 2 Driver supports the following form of a `TIME` escape clause:

```
{ts 'hh:mm:ss'}
```

Using literals or expressions as CALL statement parameters:

The DB2 Universal JDBC Driver supports the use of only parameter markers as CALL statement parameters.

The DB2 JDBC Type 2 Driver supports the use of literal and parameters as call statement parameters.

Including a CALL statement in a statement batch:

The DB2 Universal JDBC Driver supports CALL statements in a statement batch.

The DB2 UDB Type 2 Driver does not support CALL statements in a statement batch.

Removal of extra characters from SQL statement text:

The DB2 Universal JDBC Driver does not remove white-space characters, such as spaces, tabs, and new-line characters, from SQL statement text before it passes that text to the database server.

The DB2 UDB Type 2 Driver removes white-space characters from SQLstatement text before it passes that text to the database server.

Result of executing PreparedStatement.executeBatch:

When a PreparedStatement.executeBatch statement is executed under the DB2 Universal JDBC Driver, the driver returns an `int` array of update counts. Each element of the array contains the number of rows that were updated by a statement in the batch.

When a PreparedStatement.executeBatch statement is executed under the DB2 UDB Type 2 Driver, the driver cannot determine the update counts, so it returns -3 for each update count.

Support for compound SQL:

The DB2 Universal JDBC Driver driver does not support compound SQL blocks.

The DB2 UDB Type 2 Driver supports compound SQL blocks. You can use a `PreparedStatement.executeUpdate` or `Statement.executeUpdate` call to execute a compound SQL block.

Result of not setting a parameter in a batched update:

The DB2 Universal JDBC Driver driver throws an exception after a `PreparedStatement.addBatch` call if a parameter is not set.

The DB2 UDB Type 2 Driver throws an exception after the `PreparedStatement.executeBatch` call if a parameter is not set for any of the statements in the batch.

Ability to call uncatalogued stored procedures:

The DB2 Universal JDBC Driver driver does not let you call stored procedures that are not defined in the DB2 catalog.

The DB2 UDB Type 2 Driver lets you call stored procedures that are not defined in the DB2 catalog.

Related concepts:

- "Security under the DB2 Universal JDBC Driver" on page 444
- "LOBs in JDBC applications with the DB2 Universal JDBC Driver" on page 289
- "Security under the DB2 JDBC Type 2 Driver" on page 443

Related tasks:

- "Making batch updates in JDBC applications" on page 304
- "Using CallableStatement methods to call stored procedures" on page 281
- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270
- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Using the PreparedStatement.executeUpdate method to update data in DB2 tables" on page 279
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 309
- "Using the Statement.executeUpdate method to create and modify DB2 objects" on page 277

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370
- "Error codes issued by the DB2 Universal JDBC Driver" on page 434
- "SQLSTATEs issued by the DB2 Universal JDBC Driver" on page 434
- "Comparison of driver support for JDBC APIs" on page 376
- "Java, JDBC, and SQL data types" on page 365

SQLJ differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers

SQLJ support in the DB2 Universal JDBC Driver differs from SQLJ support in the other DB2 JDBC drivers in the following areas:

Differences in serialized profiles:

The DB2 JDBC Type 2 Driver and the DB2 Universal JDBC Driver produce different binary code when you execute their SQLJ translator and the SQLJ customizer utilities. Therefore, SQLJ applications that you translated and customized using the DB2 JDBC Type 2 Driver `sqlj` and `db2profc` utilities do not run under the DB2 Universal JDBC Driver. ***Before you can run those SQLJ***

applications under the DB2 Universal JDBC Driver, you must retranslate and recustomize the applications using the DB2 Universal JDBC Driver `sqlj` and `db2sqljcustomize` utilities. You must do so even if you have not modified the applications.

SQL VALUES support:

The DB2 JDBC Type 2 Driver supports the SQL VALUES statement in an SQLJ statement clause, but the DB2 Universal JDBC Driver does not. Therefore, you need to modify your SQLJ applications that include VALUES statements.

Example: Suppose that an SQLJ program contains the following statement:

```
#sql [ctxt] hv = {VALUES (MY_ROUTINE(1))};
```

For the DB2 Universal JDBC Driver, you need to change that statement to something like this:

```
#sql [ctxt] {SELECT MY_ROUTINE(1) INTO :hv FROM SYSIBM.SYSDUMMY1};
```

Compound SQL statement support:

The DB2 JDBC Type 2 Driver supports compound SQL statements in an SQLJ statement clause, but the DB2 Universal JDBC Driver does not. Therefore, you need to modify your SQLJ applications that include SQLJ statements with BEGIN COMPOUND and END COMPOUND. If you use compound statements to do batch updates, you can use the SQLJ batch update programming interfaces instead.

Difference in connection techniques:

The connection techniques that are available, and the driver names and URLs that are used for those connection techniques, vary from driver to driver. See Connect to a data source using SQLJ for more information.

Support for scrollable and updatable iterators:

SQLJ with the DB2 Universal JDBC Driver supports scrollable and updatable iterators.

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) supports scrollable cursors but not updatable iterators.

Dynamic execution of SQL statements under WebSphere Application Server:

With the DB2 Universal JDBC Driver, if you are using a version of WebSphere Application Server before version 5.0.1, all SQL statements in an SQLJ program are executed dynamically, regardless of whether you customize the SQLJ program. For WebSphere Application Server Version 5.0.1 and above, if you customize your SQLJ program, SQL statements are executed statically.

Related concepts:

- "Security under the DB2 Universal JDBC Driver" on page 444
- "LOBs in JDBC applications with the DB2 Universal JDBC Driver" on page 289
- "Security under the DB2 JDBC Type 2 Driver" on page 443

Related tasks:

- "Making batch updates in JDBC applications" on page 304

- "Using CallableStatement methods to call stored procedures" on page 281
- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270
- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Using the PreparedStatement.executeUpdate method to update data in DB2 tables" on page 279
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 309
- "Using the Statement.executeUpdate method to create and modify DB2 objects" on page 277

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370
- "Error codes issued by the DB2 Universal JDBC Driver" on page 434
- "SQLSTATEs issued by the DB2 Universal JDBC Driver" on page 434
- "Comparison of driver support for JDBC APIs" on page 376
- "Java, JDBC, and SQL data types" on page 365

Error codes issued by the DB2 Universal JDBC Driver

Error codes in the ranges +4200 to +4299, +4450 to +4499, -4200 to -4299, and -4450 to -4499 are reserved for the DB2 Universal JDBC Driver. Currently, the DB2 Universal JDBC Driver issues the following error codes:

-4200 An application that was in a global transaction in an XA environment issued an invalid commit or rollback.

-4498 A failover or failback occurred, and the transaction failed.

-4499 A fatal error occurred that resulted in a disconnect.

-99999 The DB2 Universal JDBC Driver issued an error that does not yet have an error code.

Related tasks:

- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Handling SQL errors in an SQLJ application" on page 343

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426

SQLSTATEs issued by the DB2 Universal JDBC Driver

SQLSTATEs in the range 46600 to 466ZZ are reserved for the DB2 Universal JDBC Driver. Currently, the DB2 Universal JDBC Driver returns a null SQLSTATE value for an internal error, unless the error is a DRDA error. The following SQLSTATEs are issued for DRDA errors:

08004 The application server rejected establishment of the connection.

22021 A character is not in the coded character set.

24501 The identified cursor is not open.

2D521 SQL COMMIT or ROLLBACK are invalid in the current operating environment.

58008 Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements.

58009 Execution failed due to a distribution protocol error that caused deallocation of the conversation.

58010 Execution failed due to a distribution protocol error that will affect the successful execution of subsequent DDM commands or SQL statements.

58014 The DDM command is not supported.

58015 The DDM object is not supported.

58016 The DDM parameter is not supported.

58017 The DDM parameter value is not supported.

Related tasks:

- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 282
- "Handling SQL errors in an SQLJ application" on page 343

Related reference:

- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 426

Chapter 18. Installing the JDBC drivers

The sections that follow contain information on installing the JDBC drivers.

Installing the DB2 Universal JDBC Driver

If you select JDBC support during the installation of any of the DB2 UDB for Linux, UNIX and Windows products, the installation program performs some of the installation steps for the DB2 Universal JDBC Driver. The DB2 UDB Version 8 for Windows installation program performs the following tasks:

- Installs the db2jcc.jar and sqlj.zip files and adds them to the system CLASSPATH
- Installs file db2jcct2.dll, which is required for Universal Type 2 Connectivity, in the sqllib\bin directory

The DB2 UDB Version 8 for UNIX installation program performs the following tasks:

- Installs the db2jcc.jar and sqlj.zip files
- Adds the db2jcc.jar and sqlj.zip files to the CLASSPATH statement in the db2profile (for Bourne or Korn shell) or db2cshrc (for C shell) script
- Installs file libdb2jcct2.so (libdb2jcct2.sl for HP-UX), which is required for Universal Type 2 Connectivity, in the sqllib/lib directory

You need to perform the following additional steps to install the DB2 Universal JDBC Driver.

Prerequisites:

- JDK 1.3.1 or later

  JDBC 3.0 features require a minimum level of JDK 1.4.
- TCP/IP

  Servers must be configured for TCP/IP communication in the following cases:
  - JDBC or SQLJ applications use Universal Type 4 Connectivity.
  - JDBC or SQLJ applications use Universal Type 2 Connectivity, and specify *server* and *port* in the connection URL.
- DB2 UDB for z/OS or OS/390 stored procedures

  If any JDBC or SQLJ applications will connect to a DB2 UDB for z/OS or OS/390 server, a number of stored procedures need to be installed on that server to support retrieval of DB2 catalog information, tracing, and error message formatting. The stored procedures are:
  - SQLCOLPRIVILEGES
  - SQLCOLUMNS
  - SQLFOREIGNKEYS
  - SQLGETTYPEINFO
  - SQLPRIMARYKEYS
  - SQLPROCEDURECOLS
  - SQLPROCEDURES
  - SQLSPECIALCOLUMNS
  - SQLSTATISTICS
  - SQLTABLEPRIVILEGES
  - SQLTABLES
  - SQLUDTS
  - SQLCAMESSAGE

The stored procedures are shipped with the DB2 UDB for z/OS Version 8 product. The stored procedures are shipped in the following DB2 UDB for OS/390 and z/OS Version 7 or DB2 UDB for OS/390 Version 6 PTFs:

Table 75. PTFs for DB2 for OS/390 and z/OS

DB2 for OS/390 and z/OS version	PTF number
Version 6	UQ72081 and UQ72082
Version 7	UQ72083

Ask your DB2 UDB for z/OS system administrator whether these stored procedures are installed.

- Unicode support for iSeries servers

  If any SQLJ or JDBC programs will use Universal Type 4 Connectivity to connect to a DB2 UDB for iSeries server, the OS/400 operating system must support the Unicode UTF-8 encoding scheme. The following table lists the OS/400 PTFs that you need for Unicode UTF-8 support:

Table 76. OS/400 PTFs for Unicode UTF-8 support

OS/400 version	PTF numbers
V5R3 or later	None (support is included)
V5R2	SI06541, SI06796, SI07557, SI07564, SI07565, SI07566, and SI07567
V5R1	SI06308, SI06300, SI06301, SI06302, SI06305, SI06307, and SI05872

- Java support for HP-UX clients and servers

  HP-UX servers: The DB2 Universal JDBC Driver does not support databases that are in the HP-UX default character set, Roman8. Therefore, when you create a database on an HP-UX server that you plan to access with the DB2 Universal JDBC Driver, you need to create the database with a different character set.

  HP-UX clients and servers: The Java environment on an HP-UX system requires special setup run applications under the DB2 Universal JDBC Driver.

  See Set Up the HP-UX Java Environment for details.

Procedure:

Note:

To install the DB2 Universal JDBC Driver:

1. Include db2jcc.jar in the Java application CLASSPATH, before any JAR file names for other JDBC drivers.
2. If you plan to use Kerberos security, put the following files in the Java application CLASSPATH:
   - ibmjceprovider.jar
   - ibmjcefw.jar
   - ibmjlog.jar
   - US_export_policy.jar
   - Local_policy.jar
   - ibmjgssprovider.jar
   - jaas.jar
   - ibmjceprovider.jar
   - ibmjcefw.jar
   - ibmjlog.jar

- US_export_policy.jar
- Local_policy.jar

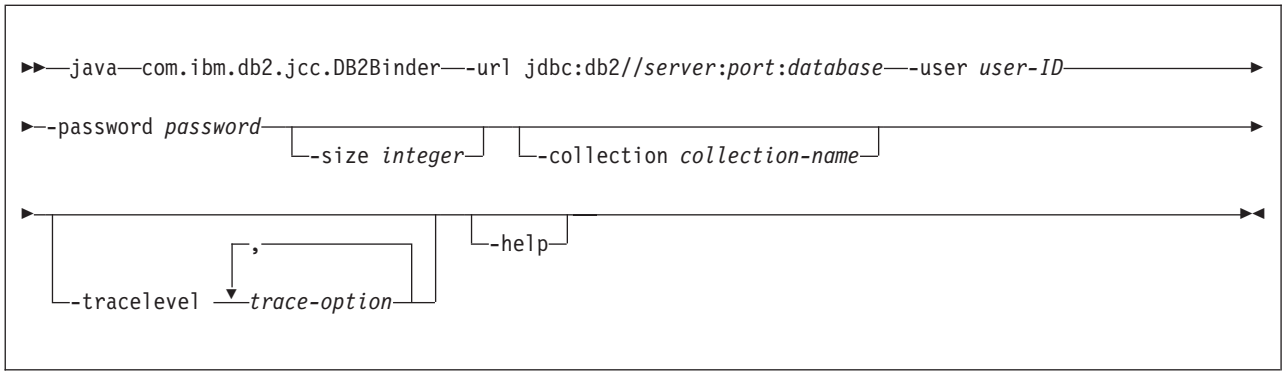
3. Include one or more license file names in the Java application CLASSPATH to permit connectivity to servers. Table 77 lists those license files.

Table 77. DB2 Universal JDBC Driver license files

License file	Server to which license file permits a connection	Product that includes license file
db2jcc_license_c.jar	Cloudscape	Cloudscape Network Server
db2jcc_license_cu.jar	Cloudscape DB2 UDB for Linux, UNIX and Windows	DB2 UDB for Linux, UNIX and Windows
db2jcc_license_cisuz.jar	Cloudscape DB2 UDB for Linux, UNIX and Windows DB2 UDB for z/OS or DB2 UDB for OS/390 and z/OS DB2 UDB for iSeries DB2 Server for VSE & VM	DB2 Connect

4. If you intend to use SQLJ, add sqlj.zip to the Java application CLASSPATH. Remove old versions of sqlj.zip and runtime.zip from the CLASSPATH.
5. If you intend to connect to a DB2 UDB for z/OS or OS/390 server, run the com.ibm.db2.jcc.DB2Binder utility to bind the DB2 packages that are used at the server by the DB2 Universal JDBC Driver.

   DB2Binder syntax:



DB2Binder option descriptions:

-url
Specifies the data source at which the JCC packages are to be bound. The variable parts of the `-url` value are:

server
The domain name or IP address of the MVS system on which the DB2 subsystem resides.

port
The TCP/IP server port number that is assigned to the DB2 subsystem. The default is 446.

database
The location name for the DB2 subsystem, as defined in the SYSIBM.LOCATIONS catalog table.

-user
Specifes the user ID under which the packages are to be bound. This user must have BIND authority on the packages.

-size
Specifies the number of DB2 packages that DB2binder binds for each of the four DB2 isolation levels and each of the two holdability values. The DB2 Universal JDBC Driver uses these packages to process dynamic SQL. In addition, the DB2binder binds a single package that the DB2 Universal JDBC Driver uses for static SQL. Therefore, the total number of packages that DB2binder binds is:

```
4*2*integer+1
```

The default value for *integer* is 3.

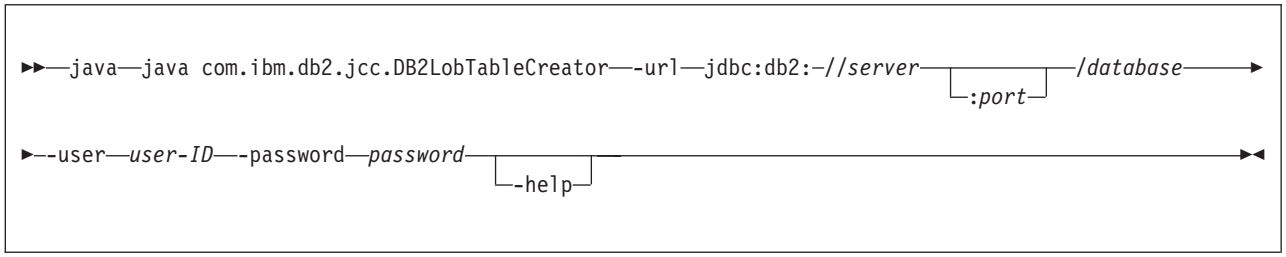
-collection
Specifies the collection ID for the packages that are used by an instance of the DB2 Universal JDBC Driver. The default is NULLID. DB2binder translates this value to uppercase.

You can create multiple instances of the JCC package set at a single location by running `com.ibm.db2.jcc.DB2Binder` multiple times, and specifying a different value for -collection each time. At run time, you select a copy of the DB2 Universal JDBC Driver by setting the `currentPackageSet` property to a value that matches a -collection value. See Properties for the DB2 Universal JDBC Driver for information on the `currentPackageSet` property.

-tracelevel
Specifies what to trace while DB2Binder runs. See the explanation of the `traceLevel` property in Properties for the DB2 Universal JDBC Driver for the options that are available.

6. If you intend to use LOB locators to access DBCLOB columns in DB2 tables on a DB2 UDB for z/OS or OS/390 server, run the com.ibm.db2.jcc.DB2LobTableCreator utility on each of those servers to create tables that are needed for fetching LOB locators.

DB2LobTableCreator syntax:



DB2LobTableCreator option descriptions:

-url
Specifies the data source at which DB2LobTableCreator is to run. The variable parts of the `-url` value are:

jdbc:db2:
Indicates that the connection is to a server in the DB2 UDB family.

server
The domain name or IP address of the database server.

port
The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

database
A name for the database server.

database is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

-user
Specifes the user ID under which DB2LobTableCreator is to run. This user must have authority to create tables in the DSNATPDB database.

-password
Specifes the password for the user ID.

-help
Specifies that the DB2LobTableCreator utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

Related tasks:

- "Setting up the HP-UX Java environment" in the *Application Development Guide: Building and Running Applications*
- "Configuring TCP/IP communications for a DB2 instance" in the *Installation and Configuration Supplement*
- "Updating the database manager configuration file on the server for TCP/IP communications" in the *Installation and Configuration Supplement*
- "Updating the services file on the server for TCP/IP communications" in the *Installation and Configuration Supplement*

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Chapter 19. JDBC and SQLJ security

The sections that follow contain information on security mechanisms that are available under the JDBC drivers.

Security under the DB2 JDBC Type 2 Driver

The DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® (DB2 JDBC Type 2 Driver) supports user ID and password security. You must set the user ID and the password, or set neither. If you do not set a user ID and password, the driver uses the user ID and password of the user who is currently logged on to the operating system.

To specify user ID and password security for a JDBC connection, use one of the following techniques.

For the `DriverManager` interface: you can specify the user ID and password directly in the `DriverManager.getConnection` invocation. For example:

```
import java.sql.*;        // JDBC base
...
String id = "db2adm";      // Set user ID
Sring pw = "db2adm";       // Set password
String url = "jdbc:db2:toronto";
                           // Set URL for the data source
Connection con = DriverManager.getConnection(url, id, pw);
                           // Create connection
```

Alternatively, you can set the user ID and password by setting the `user` and `password` properties in a `Properties` object, and then invoking the form of the `getConnection` method that includes the `Properties` object as a parameter. For example:

```
import java.sql.*;                          // JDBC base
import COM.ibm.db2.jdbc.*;                  // DB2 implementation of JDBC 2.0
...
Properties properties = new java.util.Properties();
                                            // Create Properties object
properties.put("user", "db2adm");           // Set user ID for the connection
properties.put("password", "db2adm");       // Set password for the connection
String url = "jdbc:db2:toronto";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create connection
```

For the `DataSource` interface: you can specify the user ID and password directly in the `DataSource.getConnection` invocation. For example:

```
import java.sql.*;                          // JDBC base
import COM.ibm.db2.jdbc.*;                  // DB2 implementation of JDBC 2.0
...
Context ctx=new InitialContext();           // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
                                            // Get DataSource object
String id = "db2adm";                       // Set user ID
Sring pw = "db2adm";                        // Set password
Connection con = ds.getConnection(id, pw);
                                            // Create connection
```

Alternatively, if you create and deploy the DataSource object, you can set the user ID and password by invoking the DataSource.setUser and DataSource.setPassword methods after you create the DataSource object. For example:

```
import java.sql.*;                          // JDBC base
import COM.ibm.db2.jdbc.*;                  // DB2 implementation of JDBC 2.0
...
DB2DataSource db2ds = new DB2DataSource();
                                             // Create DataSource object
db2ds.setDatabaseName("toronto");            // Set location
db2ds.setUser("db2adm");                     // Set user ID
db2ds.setPassword("db2adm");                 // Set password
```

Related concepts:

- "How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver" on page 268

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Creating and deploying DataSource objects" on page 311

Security under the DB2 Universal JDBC Driver

When you use the DB2 Universal JDBC Driver, you choose a security mechanism by specifying a value for the securityMechanism property. You can set this property in one of the following ways:

- If you use the DriverManager interface, set securityMechanism in a java.util.Properties object before you invoke the form of the getConnection method that includes the java.util.Properties parameter.
- If you use the DataSource interface, and you are creating and deploying your own DataSource objects, invoke the DataSource.setSecurityMechanism method after you create a DataSource object.

Table 78 lists the security mechanisms that the DB2 Universal JDBC Driver supports, and the value that you need to specify for the securityMechanism property to specify each security mechanism. The default security mechanism is the user ID and password mechanism.

Table 78. Security mechanisms supported by the DB2 Universal JDBC Driver

Security mechanism	securityMechanism property value
User ID and password	DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY
User ID only	DB2BaseDataSource.USER_ONLY_SECURITY
User ID and encrypted password	DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY
Encrypted user ID and encrypted password	DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY
Kerberos[1]	DB2BaseDataSource.KERBEROS_SECURITY

Note:

1. Available for Universal Type 4 Connectivity only.

Related concepts:

- "Encrypted user ID security or encrypted password security under the DB2 Universal JDBC Driver" on page 447

- "Kerberos security under the DB2 Universal JDBC Driver" on page 448
- "User ID-only security under the DB2 Universal JDBC Driver" on page 446
- "User ID and password security under the DB2 Universal JDBC Driver" on page 445

Related reference:
- "Properties for the DB2 Universal JDBC Driver" on page 370

User ID and password security under the DB2 Universal JDBC Driver

To specify user ID and password security for a JDBC connection, use one of the following techniques.

For the `DriverManager` interface: You can specify the user ID and password directly in the `DriverManager.getConnection` invocation. For example:

```
import java.sql.*;          // JDBC base
...
String id = "db2adm";       // Set user ID
String pw = "db2adm";       // Set password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, id, pw);
                            // Create connection
```

Another method is to set the user ID and password directly in the URL string. For example:

```
import java.sql.*;          // JDBC base
...
String url =
  "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose:user=db2adm;password=db2adm;";
                            // Set URL for the data source
Connection con = DriverManager.getConnection(url);
                            // Create connection
```

Alternatively, you can set the user ID and password by setting the `user` and `password` properties in a `Properties` object, and then invoking the form of the `getConnection` method that includes the `Properties` object as a parameter. Optionally, you can set the `securityMechanism` property to indicate that you are using user ID and password security. For example:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2® implementation of JDBC 2.0
...
Properties properties = new java.util.Properties();
                                            // Create Properties object
properties.put("user", "db2adm");           // Set user ID for the connection
properties.put("password", "db2adm");       // Set password for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY +
  ""));
                                            // Set security mechanism to
                                            // user ID and password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create connection
```

For the `DataSource` interface: you can specify the user ID and password directly in the `DataSource.getConnection` invocation. For example:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
Context ctx=new InitialContext();           // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
                                            // Get DataSource object
String id = "db2adm";                       // Set user ID
String pw = "db2adm";                        // Set password
Connection con = ds.getConnection(id, pw);
                                            // Create connection
```

Alternatively, if you create and deploy the DataSource object, you can set the user ID and password by invoking the DataSource.setUser and DataSource.setPassword methods after you create the DataSource object. Optionally, you can invoke the DataSource.setSecurityMechanism method property to indicate that you are using user ID and password security. For example:

```
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =    // Create DB2SimpleDataSource object
  new com.ibm.db2.jcc.DB2SimpleDataSource();
db2ds.setDriverType(4);                         // Set driver type
db2ds.setDatabaseName("san_jose");              // Set location
db2ds.setServerName("mvs1.sj.ibm.com");         // Set server name
db2ds.setPortNumber(5021);                      // Set port number
db2ds.setUser("db2adm");                        // Set user ID
db2ds.setPassword("db2adm");                    // Set password
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
                                                // Set security mechanism to
                                                // user ID and password
```

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Creating and deploying DataSource objects" on page 311
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

User ID-only security under the DB2 Universal JDBC Driver

To specify user ID security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: Set the user ID and security mechanism by setting the user and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example:

```
import java.sql.*;                           // JDBC base
import com.ibm.db2.jcc.*;                    // DB2® implementation of JDBC 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "db2adm");         // Set user ID for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY + ""));
                                          // Set security mechanism to
                                          // user ID only
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
```

```
                                        // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, you can set the user ID and security mechanism by invoking the DataSource.setUser and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```
import java.sql.*;                      // JDBC base
import com.ibm.db2.jcc.*;               // DB2 implementation of JDBC 2.0
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                        // Create DB2SimpleDataSource object
db2ds.setDriverType(4);                 // Set the driver type
db2ds.setDatabaseName("san_jose");      // Set the location
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021);              // Set the port number
db2ds.setUser("db2adm");                // Set the user ID
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY);
                                        // Set security mechanism to
                                        // user ID only
```

Encrypted user ID security or encrypted password security under the DB2 Universal JDBC Driver

If you use encrypted user ID security or encrypted password security when you access a DB2® for z/OS™ server, the Java™ Cryptography Extension, IBMJCE for z/OS needs to be enabled on the server. The Java Cryptography Extension is part of the IBM® Developer Kit for OS/390®, Java 2 Technology Edition, or the IBM Developer Kit for z/OS, Java 2 Technology Edition. For information on how to enable IBMJCE, go to this URL on the Web:

```
http://www.ibm.com/servers/eserver/zseries/software/java/aboutj2.html
```

To specify encrypted user ID or encrypted password security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: Set the user ID, password, and security mechanism by setting the user, password, and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the user ID and encrypted password security mechanism:

```
import java.sql.*;                      // JDBC base
import com.ibm.db2.jcc.*;               // DB2 implementation of JDBC 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "db2adm");       // Set user ID for the connection
properties.put("password", "db2adm");   // Set password for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY +
  ""));
                                        // Set security mechanism to
                                        // user ID and encrypted password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                        // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                        // Create the connection
```

*For the **DataSource** interface:* If you create and deploy the DataSource object, you can set the user ID, password, and security mechanism by invoking the DataSource.setUser, DataSource.setPassword, and DataSource.setSecurityMechanism methods after you create the DataSource object. For example, use code like this to set the encrypted user ID and encrypted password security mechanism:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                            // Create the DataSource object
db2ds.setDriverType(4);                     // Set the driver type
db2ds.setDatabaseName("san_jose");          // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");
                                            // Set the server name
db2ds.setPortNumber(5021);                  // Set the port number
db2ds.setUser("db2adm");                    // Set the user ID
db2ds.setPassword("db2adm");                // Set the password
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY);
                                            // Set security mechanism to
                                            // encrypted user ID and password
```

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Creating and deploying DataSource objects" on page 311
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Kerberos security under the DB2 Universal JDBC Driver

Kerberos security is available for Universal Type 4 Connectivity only.

If you use Kerberos security when you access a DB2® for z/OS™ server, you need to install and configure the following products, or their equivalents:

- The SecureWay® Security Server for z/OS and OS/390®
- OS/390 SecureWay Security Server Network Authentication and Privacy Service, which is a component of the OS/390 SecureWay Security Server

  This is the IBM® OS/390 implementation of Kerberos Version 5.

  For more information, see *OS/390 SecureWay Server Network Authentication and Privacy Service Administration*.

You also need to enable the following components of the IBM Developer Kit for OS/390, Java™ 2 Technology Edition, or the IBM Developer Kit for z/OS, Java 2 Technology Edition:

- Java Cryptography Extension (IBMJCE) for OS/390
- IBM Java Generic Security Service (IBMJGSS)
- Java Authentication and Authorization Service (JAAS) for OS/390

For information on how to enable these components, go to this URL on the Web:

`http://www.ibm.com/servers/eserver/zseries/software/java/aboutj2.html`

There are three ways to specify Kerberos security for a connection:
- With a user ID and password
- Without a user ID or password
- With a delegated credential

Using Kerberos security with a user ID and password:

For this case, Kerberos uses the specified user ID and password to obtain a ticket-granting ticket (TGT) that lets you authenticate to the DB2 server.

You need to set the `user`, `password`, `kerberosServerPrincipal`, and `securityMechanism` properties. The `kerberosServerPrincipal` property specifies the address of the Kerberos server for the realm in which the client is registered.

For the `DriverManager` interface: Set the user ID, password, Kerberos server, and security mechanism by setting the `user`, `password`, `kerberosServerPrincipal`, and `securityMechanism` properties in a `Properties` object, and then invoking the form of the `getConnection` method that includes the `Properties` object as a parameter. For example, use code like this to set the Kerberos security mechanism with a user ID and password:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "db2adm");           // Set user ID for the connection
properties.put("password", "db2adm");       // Set password for the connection
properties.put("kerberosServerPrincipal", "kdcsrv1.sj.ibm.com");
                                            // Set the Kerberos server
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + ""));
                                            // Set security mechanism to
                                            // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create the connection
```

For the `DataSource` interface: If you create and deploy the `DataSource` object, set the Kerberos server and security mechanism by invoking the `DataSource.setKerberosServerPrincipal` and `DataSource.setSecurityMechanism` methods after you create the `DataSource` object. For example:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                            // Create the DataSource object
db2ds.setDriverType(4);                     // Set the driver type
db2ds.setDatabaseName("san_jose");          // Set the location
db2ds.setUser("db2adm");                    // Set the user
db2ds.setPassword("db2adm");                // Set the password
db2ds.setServerName("mvs1.sj.ibm.com");
                                            // Set the server name
db2ds.setPortNumber(5021);                  // Set the port number
db2ds.setKerberosServerPrincipal("kdcsrv1.sj.ibm.com");
                                            // Set the Kerberos server
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
                                            // Set security mechanism to
                                            // Kerberos
```

Using Kerberos security with no user ID or password:

For this case, the Kerberos default credentials cache must contain a ticket-granting ticket (TGT) that lets you authenticate to the DB2 server.

You need to set the kerberosServerPrincipal and securityMechanism properties.

For the DriverManager interface: Set the Kerberos server and security mechanism by setting the kerberosServerPrincipal and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism without a user ID and password:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "kdcsrv1.sj.ibm.com");
                                            // Set the Kerberos server
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + ""));
                                            // Set security mechanism to
                                            // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, set the Kerberos server and security mechanism by invoking the DataSource.setKerberosServerPrincipal and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
DB2DataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
                                            // Create the DataSource object
db2ds.setDriverType(4);                     // Set the driver type
db2ds.setDatabaseName("san_jose");          // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");
                                            // Set the server name
db2ds.setPortNumber(5021);                  // Set the port number
db2ds.setKerberosServerPrincipal("kdcsrv1.sj.ibm.com");
                                            // Set the Kerberos server
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
                                            // Set security mechanism to
                                            // Kerberos
```

Using Kerberos security with a delegated credential from another principal:

For this case, you authenticate to the DB2 server using a delegated credential that another principal passes to you.

You need to set the kerberosServerPrincipal, gssCredential, and securityMechanism properties.

For the DriverManager interface: Set the Kerberos server, delegated credential, and security mechanism by setting the kerberosServerPrincipal, and securityMechanism properties in a Properties object. Because the gssCredential property is not a string, you cannot use the Properties.put method to set it.

Instead, use the DB2BaseDataSource.setGSSCredential method. Then invoke the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism without a user ID and password:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "kdcsrv1.sj.ibm.com");
                                            // Set the Kerberos server
properties.put("gssCredential",delegatedCredential);
                                            // Set the delegated credential
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + ""));

                                            // Set security mechanism to
                                            // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                            // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                            // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, set the Kerberos server, delegated credential, and security mechanism by invoking the DataSource.setKerberosServerPrincipal, DataSource.setGssCredential, and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```
DB2DataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
                                               // Create the DataSource object
db2ds.setDriverType(4);                        // Set the driver type
db2ds.setDatabaseName("san_jose");             // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");  // Set the server name
db2ds.setPortNumber(5021);                     // Set the port number
db2ds.setKerberosServerPrincipal("kdcsrv1.sj.ibm.com");
                                               // Set the Kerberos server
db2ds.setGssCredential(delegatedCredential);
                                               // Set the delegated credential
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
                                               // Set security mechanism to
                                               // Kerberos
```

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Creating and deploying DataSource objects" on page 311
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Chapter 20. Diagnosing JDBC and SQLJ problems

The sections that follow contain information on diagnosing JDBC and SQLJ problems.

Diagnosing JDBC and SQLJ problems under the DB2 Universal JDBC Driver

The sections that follow contain information on diagnosing JDBC and SQLJ problems under the DB2 Universal JDBC Driver.

JDBC and SQLJ problem diagnosis with the DB2 Universal JDBC Driver

To obtain data for diagnosing SQLJ or JDBC problems with the DB2 Universal JDBC Driver, collect trace data and run utilities that format the trace data. You should run the trace and diagnostic utilities only under the direction of IBM® software support.

If your application connects to a DB2® UDB for z/OS™ or OS/390® server, a number of stored procedures need to be installed on that server before you can collect trace data. Those stored procedures are also used for some `DatabaseMetaData` calls. The stored procedures are:

- SQLCOLPRIVILEGES
- SQLCOLUMNS
- SQLFOREIGNKEYS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS
- SQLCAMESSAGE

For DB2 UDB for OS/390 and z/OS, Version 7 or DB2 UDB for OS/390, Version 6, the stored procedures are shipped in PTFs. The PTFs are orderable through normal service channels using the following PTF numbers:

Table 79. PTFs for DB2 Universal Database for z/OS and OS/390

DB2 Universal Database for z/OS and OS/390 Version	PTF number
Version 6	UQ72081 and UQ72082
Version 7	UQ72083

Ask your DB2 UDB for z/OS system administrator whether these stored procedures are installed.

Collecting JDBC trace data:

Use one of the following procedures to start the trace:

Procedure 1:

1. If you use the `DataSource` interface to connect to a data source, invoke the `DB2BaseDataSource.setTraceLevel` method to set the type of tracing that you need. The default trace level is `TRACE_ALL`. See Properties for the DB2 Universal JDBC Driver for information on how to specify more than one type of tracing.
2. Invoke the `DB2BaseDataSource.setJccLogWriter` method to specify the trace destination and turn the trace on.

Procedure 2:

If you use the `DataSource` interface to connect to a data source, invoke the `javax.sql.DataSource.setLogWriter` method to turn the trace on. With this method, `TRACE_ALL` is the only available trace level.

If you use the `DriverManager` interface to connect to a data source, follow this procedure to start the trace.

1. Invoke the `DriverManager.getConnection` method with the `traceLevel` property set in the *info* parameter or *url* parameter for the type of tracing that you need. The default trace level is `TRACE_ALL`. See Properties for the DB2 Universal JDBC Driver for information on how to specify more than one type of tracing.
2. Invoke the `DriverManager.setLogWriter` method to specify the trace destination and turn the trace on.

After a connection is established, you can turn the trace off or back on, change the trace destination, or change the trace level with the `DB2Connection.setJccLogWriter` method. To turn the trace off, set the `logWriter` value to `null`.

The `logWriter` property is an object of type `java.io.PrintWriter`. If your application cannot handle `java.io.PrintWriter` objects, you can use the `traceFile` property to specify the destination of the trace output. To use the `traceFile` property, set the `logWriter` property to `null`, and set the `traceFile` property to the name of the file to which the driver writes the trace data. This file and the directory in which it resides must be writable. If the file already exists, the driver overwrites it.

Procedure 3: If you are using the `DriverManager` interface, specify the `traceFile` and `traceLevel` properties as part of the URL when you load the driver. For example:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" +
 ":traceFile=/u/db2p/jcctrace;" +
 "traceLevel=com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS;";
```

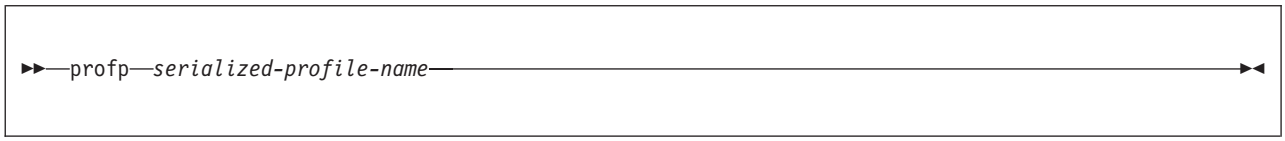
Trace example program: For a complete example of a program for tracing under the DB2 Universal JDBC Driver, see Example of tracing under the DB2 Universal JDBC Driver.

Collecting SQLJ trace data:

To collect trace data to diagnose problems during the SQLJ customization or bind process, specify the -tracelevel and -tracefile options when you run the db2sqljcustomize or db2sqljbind bind utility.

Formatting information about an SQLJ serialized profile:

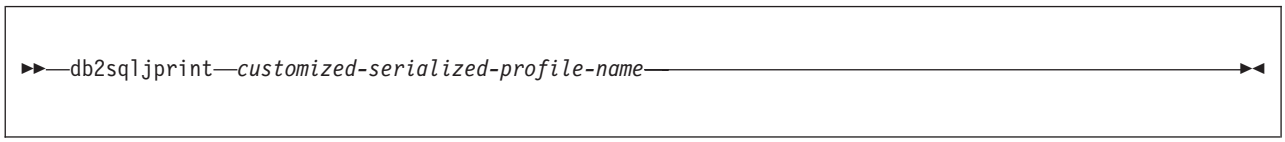
The `profp` utility formats information about each SQLJ clause in a serialized profile. The format of the `profp` utility is:



Run the `profp` utility on the serialized profile for the connection in which the error occurs. If an exception is thrown, a Java™ stack trace is generated. You can determine which serialized profile was in use when the exception was thrown from the stack trace.

Formatting information about an SQLJ customized serialized profile:

The `db2sqljprint` utility formats information about each SQLJ clause in a serialized profile that is customized for the DB2 Universal JDBC Driver. The format of the `db2sqljprint` utility is:



Run the `db2sqljprint` utility on the customized serialized profile for the connection in which the error occurs.

Related concepts:

- "Example of tracing under the DB2 Universal JDBC Driver" on page 455

Related reference:

- "db2sqljcustomize - DB2 SQLJ Profile Customizer Command" in the *Command Reference*
- "db2sqljbind - DB2 SQLJ Profile Binder Command" in the *Command Reference*
- "Properties for the DB2 Universal JDBC Driver" on page 370

Example of tracing under the DB2 Universal JDBC Driver

The following example shows a class for establishing a connection and gathering and displaying trace data under the DB2 Universal JDBC Driver. The class includes a method for the `DriverManager` interface and a method for the `DataSource` interface.

```
public class TraceExample
{

  public static void main(String[] args)
  {
    sampleConnectUsingSimpleDataSource();
    sampleConnectWithURLUsingDriverManager();
  }

  private static void sampleConnectUsingSimpleDataSource()
  {
    java.sql.Connection c = null;
    java.io.PrintWriter printWriter =
     new java.io.PrintWriter(System.out, true);
                                      // Prints to console, true means
                                      // auto-flush so you don't lose trace
    try {
      javax.sql.DataSource ds =
       new com.ibm.db2.jcc.DB2SimpleDataSource();
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(5021);
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("san_jose");
      ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);

     ds.setLogWriter(printWriter);    // This turns on tracing

     // Refine the level of tracing detail
     ((com.ibm.db2.jcc.DB2BaseDataSource) ds).
      setTraceLevel(com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_CONNECTS |
      com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_DRDA_FLOWS);

     // This connection request is traced using trace level
     // TRACE_CONNECTS | TRACE_DRDA_FLOWS
     c = ds.getConnection("myname", "mypass");

     // Change the trace level to TRACE_ALL
     // for all subsequent requests on the connection
     ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
      com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
```

Figure 60. Example of tracing under the DB2 Universal JDBC Driver (Part 1 of 5)

```
   // The following INSERT is traced using trace level TRACE_ALL
   java.sql.Statement s1 = c.createStatement();
   s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
   s1.close();

   // This code disables all tracing on the connection
   ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

   // The following INSERT statement is not traced
   java.sql.Statement s2 = c.createStatement();
   s2.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
   s2.close();

   c.close();
  }
  catch(java.sql.SQLException e) {
   com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e,
    printWriter, "[TraceExample]");
  }
  finally {
    cleanup(c, printWriter);
    printWriter.flush();
  }
}

// If the code ran successfully, the connection should
// already be closed. Check whether the connection is closed.
// If so, just return.
// If a failure occurred, try to roll back and close the connection.

private static void cleanup(java.sql.Connection c,
 java.io.PrintWriter printWriter)
{
  if(c == null) return;

  try {
   if(c.isClosed()) {
      printWriter.println("[TraceExample] " +
       "The connection was successfully closed");
      return;
   }

   // If we get to here, something has gone wrong.
   // Roll back and close the connection.
   printWriter.println("[TraceExample] Rolling back the connection");
   try {
     c.rollback();
   }
```

Figure 60. Example of tracing under the DB2 Universal JDBC Driver (Part 2 of 5)

```
    catch(java.sql.SQLException e) {
      printWriter.println("[TraceExample] " +
       "Trapped the following java.sql.SQLException while trying to roll back:");
      com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
       "[TraceExample]");
      printWriter.println("[TraceExample] " +
       "Unable to roll back the connection");
     }
     catch(java.lang.Throwable e) {
       printWriter.println("[TraceExample] Trapped the " +
        "following java.lang.Throwable while trying to roll back:");
       com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e,
        printWriter, "[TraceExample]");
       printWriter.println("[TraceExample] Unable to " +
        "roll back the connection");
     }

     // Close the connection
     printWriter.println("[TraceExample] Closing the connection");
     try {
       c.close();
     }
     catch(java.sql.SQLException e) {
       printWriter.println("[TraceExample] Exception while " +
        "trying to close the connection");
       printWriter.println("[TraceExample] Deadlocks could " +
        "occur if the connection is not closed.");
       com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
        "[TraceExample]");
     }
     catch(java.lang.Throwable e) {
       printWriter.println("[TraceExample] Throwable caught " +
        "while trying to close the connection");
       printWriter.println("[TraceExample] Deadlocks could " +
        "occur if the connection is not closed.");
       com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
        "[TraceExample]");
     }
   }
   catch(java.lang.Throwable e) {
     printWriter.println("[TraceExample] Unable to " +
      "force the connection to close");
     printWriter.println("[TraceExample] Deadlocks " +
      "could occur if the connection is not closed.");
     com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
      "[TraceExample]");
   }
 }
```

Figure 60. Example of tracing under the DB2 Universal JDBC Driver (Part 3 of 5)

```
private static void sampleConnectWithURLUsingDriverManager()
{
  java.sql.Connection c = null;

  // This time, send the printWriter to a file.
  java.io.PrintWriter printWriter = null;
  try {
    printWriter =
      new java.io.PrintWriter(
        new java.io.BufferedOutputStream(
          new java.io.FileOutputStream("/temp/driverLog.txt"), 4096), true);
  }
  catch(java.io.FileNotFoundException e) {
    java.lang.System.err.println("Unable to establish a print writer for trace");
    java.lang.System.err.flush();
    return;
  }

  try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
  }
  catch(ClassNotFoundException e) {
    printWriter.println("[TraceExample] Universal Type 4 Connectivity " +
     "is not in the application classpath. Unable to load driver.");
    printWriter.flush();
    return;
  }

  // This URL describes the target data source for Type 4 connectivity.
  // The traceLevel property is established through the URL syntax,
  // and driver tracing is directed to file "/temp/driverLog.txt"
  String databaseURL =
   "jdbc:db2://sysmvs1.stl.ibm.com:5021" +
   "/sample:traceFile=/temp/driverLog.txt;traceLevel=" +
   "(com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS " +
   "| com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS);";

  // Set other properties
  java.util.Properties properties = new java.util.Properties();
  properties.setProperty("user", "myname");
  properties.setProperty("password", "mypass");
```

Figure 60. Example of tracing under the DB2 Universal JDBC Driver (Part 4 of 5)

```
    try {
     // This connection request is traced using trace level
     // TRACE_CONNECTS | TRACE_DRDA_FLOWS
     c = java.sql.DriverManager.getConnection(databaseURL, properties);

     // Change the trace level for all subsequent requests
     // on the connection to TRACE_ALL
     ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
      com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);

     // The following INSERT is traced using trace level TRACE_ALL
     java.sql.Statement s1 = c.createStatement();
     s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
     s1.close();

     // Disable all tracing on the connection
     ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

      // The following SQL insert code is not traced
      java.sql.Statement s2 = c.createStatement();
      s2.executeUpdate("insert into sampleTable(sampleColumn) values(1)");
      s2.close();

      c.close();
    }
    catch(java.sql.SQLException e) {
     com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
      "[TraceExample]");
    }
    finally {
      cleanup(c, printWriter);
      printWriter.flush();
    }
  }
}
```

Figure 60. Example of tracing under the DB2 Universal JDBC Driver (Part 5 of 5)

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 272
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 270

Related reference:

- "Properties for the DB2 Universal JDBC Driver" on page 370

Diagnosing JDBC and SQLJ problems under the DB2 JDBC Type 2 Driver

The sections that follow contain information on diagnosing JDBC and SQLJ problems under the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver).

CLI/ODBC/JDBC trace facility

This topic discusses the following subjects:

- "DB2 CLI and DB2 JDBC trace configuration" on page 461
- "DB2 CLI trace options and the db2cli.ini file" on page 462
- "DB2 JDBC trace options and the db2cli.ini file" on page 463

The DB2 CLI and the CLI-based Legacy Type 2 JDBC Driver for Linux, UNIX®, and Windows® offer comprehensive tracing facilities. By default, these facilities are disabled and use no additional computing resources. When enabled, the trace facilities generate one or more text log files whenever an application accesses the appropriate driver (DB2 CLI or CLI-based Legacy Type 2 JDBC Driver). These log files provide detailed information about:

- the order in which CLI or JDBC functions were called by the application
- the contents of input and output parameters passed to and received from CLI or JDBC functions
- the return codes and any error or warning messages generated by CLI or JDBC functions

DB2 CLI and DB2® JDBC trace file analysis can benefit application developers in a number of ways. First, subtle program logic and parameter initialization errors are often evident in the traces. Second, DB2 CLI and DB2 JDBC traces may suggest ways of better tuning an application or the databases it accesses. For example, if a DB2 CLI trace shows a table being queried many times on a particular set of attributes, an index corresponding to those attributes might be created on the table to improve application performance. Finally, analysis of DB2 CLI and DB2 JDBC trace files can help application developers understand how a third party application or interface is behaving.

DB2 CLI and DB2 JDBC trace configuration:

The configuration parameters for both DB2 CLI and DB2 JDBC traces facilities are read from the DB2 CLI configuration file `db2cli.ini`. By default, this file is located in the `\sqllib` path on the Windows platform and the `/sqllib/cfg` path on UNIX platforms. You can override the default path by setting the DB2CLIINIPATH environment variable. On the Windows platform, an additional `db2cli.ini` file may be found in the user's profile (or home) directory if there are any user-defined data sources defined using the ODBC Driver Manager. This `db2cli.ini` file will override the default file.

To view the current `db2cli.ini` trace configuration parameters from the command line processor, issue the following command:

```
db2 GET CLI CFG FOR SECTION COMMON
```

There are three ways to modify the `db2cli.ini` file to configure the DB2 CLI and DB2 JDBC trace facilities:

- use the DB2 Configuration Assistant if it is available
- manually edit the `db2cli.ini` file using a text editor
- issue the UPDATE CLI CFG command from the command line processor

For example, the following command issued from the command line processor updates the `db2cli.ini` file and enables the JDBC tracing facility:

```
db2 UPDATE CLI CFG FOR SECTION COMMON USING jdbctrace 1
```

Notes:

1. Typically the DB2 CLI and DB2 JDBC trace configuration options are only read from the `db2cli.ini` configuration file at the time an application is initialized. However, a special `db2cli.ini` trace option, TraceRefreshInterval, can be used to indicate an interval at which specific DB2 CLI trace options are reread from the `db2cli.ini` file.
2. The DB2 CLI tracing facility can also be configured dynamically by setting the SQL_ATTR_TRACE and SQL_ATTR_TRACEFILE environment attributes. These settings will override the settings contained in the `db2cli.ini` file.

Important: Disable the DB2 CLI and DB2 JDBC trace facilities when they are not needed. Unnecessary tracing can reduce application performance and may generate unwanted trace log files. DB2 does not delete any generated trace files and will append new trace information to any existing trace files.

DB2 CLI Trace options and the db2cli.ini file:

When an application using the DB2 CLI driver begins execution, the driver checks for trace facility options in the [COMMON] section of the db2cli.ini file. These trace options are specific trace keywords that are set to certain values in the `db2cli.ini` file under the [COMMON] section.

Note: Because DB2 CLI trace keywords appear in the [COMMON] section of the `db2cli.ini` file, their values apply to all database connections through the DB2 CLI driver.

The DB2 CLI trace keywords that can be defined are:

- Trace
- TraceComm
- TraceErrImmediate
- TraceFileName
- TraceFlush
- TraceFlushOnError
- TraceLocks
- TracePathName
- TracePIDList
- TracePIDTID
- TraceRefreshInterval
- TraceStmtOnly
- TraceTime
- TraceTimeStamp

Note: DB2 CLI trace keywords are only read from the `db2cli.ini` file once at application initialization time unless the TraceRefreshInterval keyword is set. If this keyword is set, the Trace and TracePIDList keywords are reread from the `db2cli.ini` file at the specified interval and applied, as appropriate, to the currently executing application.

An example `db2cli.ini` file trace configuration using these DB2 CLI keywords and values is:

```
[COMMON]
trace=1
TraceFileName=\temp\clitrace.txt
TraceFlush=1
```

Notes:

1. CLI trace keywords are NOT case sensitive. However, path and file name keyword values may be case-sensitive on some operating systems (such as UNIX).
2. If either a DB2 CLI trace keyword or its associated value in the `db2cli.ini` file is invalid, the DB2 CLI trace facility will ignore it and use the default value for that trace keyword instead.

DB2 JDBC Trace options and the db2cli.ini file:

When an application using the CLI-based Legacy Type 2 JDBC Driver begins execution, the driver also checks for trace facility options in the db2cli.ini file. As with the DB2 CLI trace options, DB2 JDBC trace options are specified as keyword/value pairs located under the [COMMON] section of the `db2cli.ini` file.

Note: Because DB2 JDBC trace keywords appear in the [COMMON] section of the `db2cli.ini` file, their values apply to all database connections through the CLI-based Legacy Type 2 JDBC Driver.

The DB2 JDBC trace keywords that can be defined are:

- JDBCTrace
- JDBCTracePathName
- JDBCTraceFlush

JDBCTrace = 0 | 1

The JDBCTrace keyword controls whether or not other DB2 JDBC tracing keywords have any effect on program execution. Setting JDBCTrace to its default value of 0 disables the DB2 JDBC trace facility. Setting JDBCTrace to 1 enables it.

By itself, the JDBCTrace keyword has little effect and produces no trace output unless the JDBCTracePathName keyword is also specified.

JDBCTracePathName = <fully_qualified_trace_path_name>

The value of JDBCTracePathName is the fully qualified path of the directory to which all DB2 JDBC trace information is written. The DB2 JDBC trace facility attempts to generate a new trace log file each time a JDBC application is executed using the CLI-based Legacy Type 2 JDBC Driver. If the application is multithreaded, a separate trace log file will be generated for each thread. A concatenation of the application process ID, the thread sequence number, and a thread-identifying string are automatically used to name trace log files. There is no default path name to which DB2 JDBC trace output log files are written.

JDBCTraceFlush = 0 | 1

The JDBCTraceFlush keyword specifies how often trace information is written to the DB2 JDBC trace log file. By default, JDBCTraceFlush is set to 0 and each DB2 JDBC trace log file is kept open until the traced application or thread terminates normally. If the application terminates abnormally, some trace information that was not written to the trace log file may be lost.

To ensure the integrity and completeness of the trace information written to the DB2 JDBC trace log file, the JDBCTraceFlush keyword can be set to 1. After each trace entry has been written to the trace log file, the DB2 JDBC driver closes the file and then reopens it, appending new trace entries to the end of the file. This guarantees that no trace information will be lost.

Note: *Each DB2 JDBC log file close and reopen operation incurs significant input/output overhead and can reduce application performance considerably*.

An example `db2cli.ini` file trace configuration using these DB2 JDBC keywords and values is:

```
[COMMON]
jdbctrace=1
JdbcTracePathName=\temp\jdbctrace\
JDBCTraceFlush=1
```

Notes:

1. JDBC trace keywords are NOT case sensitive. However, path and file name keyword values may be case-sensitive on some operating systems (such as UNIX).
2. If either a DB2 JDBC trace keyword or its associated value in the `db2cli.ini` file is invalid, the DB2 JDBC trace facility will ignore it and use the default value for that trace keyword instead.
3. Enabling DB2 JDBC tracing does not enable DB2 CLI tracing. The CLI-based Legacy Type 2 JDBC Driver depends on the DB2 CLI driver to access the database. Consequently, Java™ developers may also want to enable DB2 CLI tracing for additional information on how their applications interact with the database through the various software layers. DB2 JDBC and DB2 CLI trace options are independent of each other and can be specified together in any order under the [COMMON] section of the `db2cli.ini` file.

DB2 CLI Driver trace versus ODBC driver manager trace:

It is important to understand the difference between an ODBC driver manager trace and a DB2 CLI driver trace. An ODBC driver manager trace shows the ODBC function calls made by an ODBC application to the ODBC driver manager. In contrast, a DB2 CLI driver trace shows the function calls made by the ODBC driver manager to the DB2 CLI driver *on behalf of the application*.

An ODBC driver manager might forward some function calls directly from the application to the DB2 CLI driver. However, the ODBC driver manager might also delay or avoid forwarding some function calls to the driver. The ODBC driver manager may also modify application function arguments or map application functions to other functions before forwarding the call on to the DB2 CLI driver.

Reasons for application function call intervention by the ODBC driver manager include:

- Applications written using ODBC 2.0 functions that have been deprecated in ODBC 3.0 will have the old functions mapped to new functions.
- ODBC 2.0 function arguments deprecated in ODBC 3.0 will be mapped to equivalent ODBC 3.0 arguments.

- The Microsoft® cursor library will map calls such as `SQLExtendedFetch()` to multiple calls to `SQLFetch()` and other supporting functions to achieve the same end result.
- ODBC driver manager connection pooling will usually defer `SQLDisconnect()` requests (or avoid them altogether if the connection gets reused).

For these and other reasons, application developers may find an ODBC driver manager trace to be a useful complement to the DB2 CLI driver trace.

For more information on capturing and interpreting ODBC driver manager traces, refer to the ODBC driver manager documentation. On the Windows platforms, refer to the Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference, also available online at: http://www.msdn.microsoft.com/.

DB2 CLI Driver, CLI-based Legacy Type 2 JDBC Driver, and DB2 traces:

Internally, the CLI-based Legacy Type 2 JDBC Driver makes use of the DB2 CLI driver for database access. For example, the Java getConnection() method is internally mapped by the CLI-based Legacy Type 2 JDBC Driver to the DB2 CLI `SQLConnect()` function. As a result, Java developers might find a DB2 CLI trace to be a useful complement to the DB2 JDBC trace.

The DB2 CLI driver makes use of many internal and DB2 specific functions to do its work. These internal and DB2 specific function calls are logged in the DB2 trace. Application developers will not find DB2 traces useful, as they are only meant to assist IBM® Service in problem determination and resolution.

DB2 CLI and DB2 JDBC traces and CLI or Java stored procedures:

On all workstation platforms, the DB2 CLI and DB2 JDBC trace facilities can be used to trace DB2 CLI and DB2 JDBC stored procedures.

Most of the DB2 CLI and DB2 JDBC trace information and instructions given in earlier sections is generic and applies to both applications and stored procedures equally. However, unlike applications which are clients of a database server (and typically execute on a machine separate from the database server), stored procedures execute at the database server. Therefore, the following additional steps must be taken when tracing DB2 CLI or DB2 JDBC stored procedures:

- Ensure the trace keyword options are specified in the `db2cli.ini` file located at the DB2 server.
- If the TraceRefreshInterval keyword is not set to a positive, non-zero value, ensure all keywords are configured correctly prior to database startup time (that is, when the db2start command is issued). Changing trace settings while the database server is running may have unpredictable results. For example, if the TracePathName is changed while the server is running, then the next time a stored procedure is executed, some trace files may be written to the new path, while others are written to the original path. To ensure consistency, restart the server any time a trace keyword other than Trace or TracePIDList is modified.

Related concepts:

- "db2cli.ini initialization file" in the *CLI Guide and Reference, Volume 1*
- "CLI and JDBC trace files" on page 466

Related reference:

- "SQLSetEnvAttr function (CLI) - Set environment attribute" in the *CLI Guide and Reference, Volume 2*
- "db2trc - Trace Command" in the *Command Reference*
- "GET CLI CONFIGURATION Command" in the *Command Reference*
- "UPDATE CLI CONFIGURATION Command" in the *Command Reference*
- "Miscellaneous variables" in the *Administration Guide: Performance*
- "CLI/ODBC configuration keywords listing by category" in the *CLI Guide and Reference, Volume 1*

CLI and JDBC trace files

Applications that access the DB2® CLI and DB2 JDBC drivers can make use of the DB2 CLI and DB2 JDBC trace facilities. These utilities record all function calls made by the DB2 CLI or DB2 JDBC drivers to a log file which is useful for problem determination. This topic discusses how to access and interpret these log files generated by the tracing facilities:

- "CLI and JDBC trace file location"
- "CLI trace file interpretation" on page 467
- "JDBC trace file interpretation" on page 471

CLI and JDBC trace file location:

If the TraceFileName keyword was used in the `db2cli.ini` file to specify a fully qualified file name, then the DB2 CLI trace log file will be in the location specified. If a relative file name was specified for the DB2 CLI trace log file name, the location of that file will depend on what the operating system considers to be the current path of the application.

Note: If the user executing the application does not have sufficient authority to write to the trace log file in the specified path, no file will be generated and no warning or error is given.

If either or both of the TracePathName and JDBCTracePathName keywords were used in the `db2cli.ini` file to specify fully qualified directories, then the DB2 CLI and DB2 JDBC trace log files will be in the location specified. If a relative directory name was specified for either or both trace directories, the operating system will determine its location based on what it considers to be the current path of the application.

Note: If the user executing the application does not have sufficient authority to write trace files in the specified path, no file will be generated and no warning or error is given. If the specified trace path does not exist, it will not be created.

The DB2 CLI and DB2 JDBC trace facilities automatically use the application's process ID and thread sequence number to name the trace log files when the TracePathName and JDBCTracePathName keywords have been set. For example, a DB2 CLI trace of an application with three threads might generate the following DB2 CLI trace log files: `100390.0`, `100390.1`, `100390.2`.

Similarly, a DB2 JDBC trace of a Java™ application with two threads might generate the following JDBC trace log files: `7960main.trc`, `7960Thread-1.trc`.

Note: If the trace directory contains both old and new trace log files, file date and time stamp information can be used to locate the most recent trace files.

If no DB2 CLI or DB2 JDBC trace output files appear to have been created:

- Verify that the trace configuration keywords are set correctly in the `db2cli.ini` file. Issuing the `db2 GET CLI CFG FOR SECTION COMMON` command from the command line processor is a quick way to do this.
- Ensure the application is restarted after updating the `db2cli.ini` file. Specifically, the DB2 CLI and DB2 JDBC trace facilities are initialized during application startup. Once initialized, the DB2 JDBC trace facility cannot be reconfigured. The DB2 CLI trace facility can be reconfigured at run time but only if the TraceRefreshInterval keyword was appropriately specified prior to application startup.

  Note: Only the Trace and TracePIDList DB2 CLI keywords can be reconfigured at run time. *Changes made to other DB2 CLI keywords, including TraceRefreshInterval, have no effect without an application restart.*
- If the TraceRefreshInterval keyword was specified prior to application startup, and if the Trace keyword was initially set to 0, ensure that enough time has elapsed for the DB2 CLI trace facility to reread the Trace keyword value.
- If either or both the TracePathName and JDBCTracePathName keywords are used to specify trace directories, ensure those directories exist prior to starting the application.
- Ensure the application has write access to the specified trace log file or trace directory.
- Check the DB2CLIINIPATH environment variable. If set, the DB2 CLI and DB2 JDBC trace facilities expect the `db2cli.ini` file to be at the location specified by this variable.
- If the application uses ODBC to interface with the DB2 CLI driver, verify that one of the `SQLConnect()`, `SQLDriverConnect()` or `SQLBrowseConnect()` functions have been successfully called. No entries will be written to the DB2 CLI trace log files until a database connection has successfully been made.

CLI trace file interpretation:

DB2 CLI traces always begin with a header that identifies the process ID and thread ID of the application that generated the trace, the time the trace began, and product specific information such as the local DB2 build level and DB2 CLI driver version. For example:

```
1   [ Process: 1227, Thread: 1024 ]
2   [ Date, Time:           01-27-2002 13:46:07.535211 ]
3   [ Product:              QDB2/LINUX 7.1.0 ]
4   [ Level Identifier:     02010105 ]
5   [ CLI Driver Version:   07.01.0000 ]
6   [ Informational Tokens: "DB2 v7.1.0","n000510","" ]
```

Note: Trace examples used in this section have line numbers added to the left hand side of the trace. These line numbers have been added to aid the discussion and will *not* appear in an actual DB2 CLI trace.

Immediately following the trace header, there are usually a number of trace entries related to environment and connection handle allocation and initialization. For example:

```
7   SQLAllocEnv( phEnv=&bffff684 )
8       ----> Time elapsed - +9.200000E-004 seconds

9   SQLAllocEnv( phEnv=0:1 )
10      <---- SQL_SUCCESS   Time elapsed - +7.500000E-004 seconds

11  SQLAllocConnect( hEnv=0:1, phDbc=&bffff680 )
12      ----> Time elapsed - +2.334000E-003 seconds

13  SQLAllocConnect( phDbc=0:1 )
14      <---- SQL_SUCCESS   Time elapsed - +5.280000E-004 seconds

15  SQLSetConnectOption( hDbc=0:1, fOption=SQL_ATTR_AUTOCOMMIT, vParam=0 )
16      ----> Time elapsed - +2.301000E-003 seconds

17  SQLSetConnectOption( )
18      <---- SQL_SUCCESS   Time elapsed - +3.150000E-004 seconds

19  SQLConnect( hDbc=0:1, szDSN="SAMPLE", cbDSN=-3, szUID="", cbUID=-3,
                          szAuthStr="", cbAuthStr=-3 )
20      ----> Time elapsed - +7.000000E-005 seconds
21  ( DBMS NAME="DB2/LINUX", Version="07.01.0000", Fixpack="0x22010105" )

22  SQLConnect( )
23      <---- SQL_SUCCESS   Time elapsed - +5.209880E-001 seconds
24  ( DSN=""SAMPLE"" )

25  ( UID=" " )

26  ( PWD="*" )
```

In the above trace example, notice that there are two entries for each DB2 CLI function call (for example, lines 19-21 and 22-26 for the `SQLConnect()` function call). This is always the case in DB2 CLI traces. The first entry shows the input parameter values passed to the function call while the second entry shows the function output parameter values and return code returned to the application.

The above trace example shows that the `SQLAllocEnv()` function successfully allocated an environment handle (phEnv=0:1) at line 9. That handle was then passed to the `SQLAllocConnect()` function which successfully allocated a database connection handle (phDbc=0:1) as of line 13. Next, the `SQLSetConnectOption()` function was used to set the phDbc=0:1 connection's SQL_ATTR_AUTOCOMMIT attribute to SQL_AUTOCOMMIT_OFF (vParam=0) at line 15. Finally, `SQLConnect()` was called to connect to the target database (SAMPLE) at line 19.

Included in the input trace entry of the `SQLConnect()` function on line 21 is the build and FixPak level of the target database server. Other information that might also appear in this trace entry includes input connection string keywords and the code pages of the client and server. For example, suppose the following information also appeared in the `SQLConnect()` trace entry:

```
( Application Codepage=819, Database  Codepage=819,
  Char Send/Recv Codepage=819, Graphic Send/Recv Codepage=819,
  Application Char Codepage=819, Application Graphic Codepage=819 )
```

This would mean the application and the database server were using the same code page (819).

The return trace entry of the `SQLConnect()` function also contains important connection information (lines 24-26 in the above example trace). Additional information that might be displayed in the return entry includes any PATCH1 or PATCH2 keyword values that apply to the connection. For example, if

PATCH2=27,28 was specified in the db2cli.ini file under the COMMON section, the following line should also appear in the SQLConnect() return entry:

```
( PATCH2="27,28" )
```

Following the environment and connection related trace entries are the statement related trace entries. For example:

```
27  SQLAllocStmt( hDbc=0:1, phStmt=&bffff684 )
28      ---> Time elapsed - +1.868000E-003 seconds

29  SQLAllocStmt( phStmt=1:1 )
30      <--- SQL_SUCCESS   Time elapsed - +6.890000E-004 seconds

31  SQLExecDirect( hStmt=1:1, pszSqlStr="CREATE TABLE GREETING (MSG
                                  VARCHAR(10))", cbSqlStr=-3 )
32      ---> Time elapsed - +2.863000E-003 seconds
33  ( StmtOut="CREATE TABLE GREETING (MSG VARCHAR(10))" )

34  SQLExecDirect( )
35      <--- SQL_SUCCESS   Time elapsed - +2.387800E-002 seconds
```

In the above trace example, the database connection handle (phDbc=0:1) was used to allocate a statement handle (phStmt=1:1) at line 29. An unprepared SQL statement was then executed on that statement handle at line 31. If the TraceComm=1 keyword had been set in the db2cli.ini file, the SQLExecDirect() function call trace entries would have shown additional client-server communication information as follows:

```
SQLExecDirect( hStmt=1:1, pszSqlStr="CREATE TABLE GREETING (MSG
                             VARCHAR(10))", cbSqlStr=-3 )
    ---> Time elapsed - +2.876000E-003 seconds
( StmtOut="CREATE TABLE GREETING (MSG VARCHAR(10))" )

    sqlccsend( ulBytes - 232 )
    sqlccsend( Handle - 1084869448 )
    sqlccsend( ) - rc - 0, time elapsed - +1.150000E-004
    sqlccrecv( )
    sqlccrecv( ulBytes - 163 ) - rc - 0, time elapsed - +2.243800E-002

SQLExecDirect( )
    <--- SQL_SUCCESS   Time elapsed - +2.384900E-002 seconds
```

Notice the additional sqlccsend() and sqlccrecv() function call information in this trace entry. The sqlccsend() call information reveals how much data was sent from the client to the server, how long the transmission took, and the success of that transmission (0 = SQL_SUCCESS). The sqlccrecv() call information then reveals how long the client waited for a response from the server and the amount of data included in the response.

Often, multiple statement handles will appear in the DB2 CLI trace. By paying close attention to the statement handle identifier, one can easily follow the execution path of a statement handle independent of all other statement handles appearing in the trace.

Statement execution paths appearing in the DB2 CLI trace are usually more complicated than the example shown above. For example:

```
36  SQLAllocStmt( hDbc=0:1, phStmt=&bffff684 )
37      ---> Time elapsed - +1.532000E-003 seconds

38  SQLAllocStmt( phStmt=1:2 )
39      <--- SQL_SUCCESS   Time elapsed - +6.820000E-004 seconds
```

```
40  SQLPrepare( hStmt=1:2, pszSqlStr="INSERT INTO GREETING VALUES ( ? )",
                cbSqlStr=-3 )
41      ---> Time elapsed - +2.733000E-003 seconds
42  ( StmtOut="INSERT INTO GREETING VALUES ( ? )" )

43  SQLPrepare( )
44      <--- SQL_SUCCESS   Time elapsed - +9.150000E-004 seconds

45  SQLBindParameter( hStmt=1:2, iPar=1, fParamType=SQL_PARAM_INPUT,
                      fCType=SQL_C_CHAR, fSQLType=SQL_CHAR, cbColDef=14,
                      ibScale=0, rgbValue=&080eca70, cbValueMax=15,
                      pcbValue=&080eca4c )
46      ---> Time elapsed - +4.091000E-003 seconds

47  SQLBindParameter( )
48      <--- SQL_SUCCESS   Time elapsed - +6.780000E-004 seconds

49  SQLExecute( hStmt=1:2 )
50      ---> Time elapsed - +1.337000E-003 seconds
51  ( iPar=1, fCType=SQL_C_CHAR, rgbValue="Hello World!!!", pcbValue=14,
      piIndicatorPtr=14 )

52  SQLExecute( )
53      <--- SQL_ERROR   Time elapsed - +5.951000E-003 seconds
```

In the above trace example, the database connection handle (phDbc=0:1) was used to allocate a second statement handle (phStmt=1:2) at line 38. An SQL statement with one parameter marker was then prepared on that statement handle at line 40. Next, an input parameter (iPar=1) of the appropriate SQL type (SQL_CHAR) was bound to the parameter marker at line 45. Finally, the statement was executed at line 49. Notice that both the contents and length of the input parameter (rgbValue=″Hello World!!!″, pcbValue=14) are displayed in the trace on line 51.

The `SQLExecute()` function fails at line 52. If the application calls a diagnostic DB2 CLI function like `SQLError()` to diagnose the cause of the failure, then that cause will appear in the trace. For example:

```
54  SQLError( hEnv=0:1, hDbc=0:1, hStmt=1:2, pszSqlState=&bffff680,
              pfNativeError=&bfffee78, pszErrorMsg=&bffff280,
              cbErrorMsgMax=1024, pcbErrorMsg=&bfffee76 )
55      ---> Time elapsed - +1.512000E-003 seconds

56  SQLError( pszSqlState="22001", pfNativeError=-302, pszErrorMsg="[IBM][CLI
       Driver][DB2/LINUX] SQL0302N  The value of a host variable in the EXECUTE
       or OPEN statement is too large for its corresponding use.
       SQLSTATE=22001", pcbErrorMsg=157 )
57      <--- SQL_SUCCESS   Time elapsed - +8.060000E-004 seconds
```

The error message returned at line 56 contains the DB2 native error code that was generated (SQL0302N), the sqlstate that corresponds to that code (SQLSTATE=22001) and a brief description of the error. In this example, the source of the error is evident: on line 49, the application is trying to insert a string with 14 characters into a column defined as VARCHAR(10) on line 31.

If the application does not respond to a DB2 CLI function warning or error return code by calling a diagnostic function like `SQLError()`, the warning or error message should still be written to the DB2 CLI trace. However, the location of that message in the trace may not be close to where the error actually occurred. Furthermore, the trace will indicate that the error or warning message was not retrieved by the

application. For example, if not retrieved, the error message in the above example might not appear until a later, seemingly unrelated DB2 CLI function call as follows:

```
SQLDisconnect( hDbc=0:1 )
    ---> Time elapsed - +1.501000E-003 seconds
    sqlccsend( ulBytes - 72 )
    sqlccsend( Handle - 1084869448 )
    sqlccsend( ) - rc - 0, time elapsed - +1.080000E-004
    sqlccrecv( )
    sqlccrecv( ulBytes - 27 ) - rc - 0, time elapsed - +1.717950E-001
( Unretrieved error message="SQL0302N  The value of a host variable in the
  EXECUTE or OPEN statement is too large for its corresponding use.
  SQLSTATE=22001" )

SQLDisconnect( )
    <--- SQL_SUCCESS   Time elapsed - +1.734130E-001 seconds
```

The final part of a DB2 CLI trace should show the application releasing the database connection and environment handles that it allocated earlier in the trace. For example:

```
58  SQLTransact( hEnv=0:1, hDbc=0:1, fType=SQL_ROLLBACK )
59      ---> Time elapsed - +6.085000E-003 seconds
60  ( ROLLBACK=0 )

61  SQLTransact( )
        <--- SQL_SUCCESS   Time elapsed - +2.220750E-001 seconds

62  SQLDisconnect( hDbc=0:1 )
63      ---> Time elapsed - +1.511000E-003 seconds

64  SQLDisconnect( )
65      <--- SQL_SUCCESS   Time elapsed - +1.531340E-001 seconds

66  SQLFreeConnect( hDbc=0:1 )
67      ---> Time elapsed - +2.389000E-003 seconds

68  SQLFreeConnect( )
69      <--- SQL_SUCCESS   Time elapsed - +3.140000E-004 seconds

70  SQLFreeEnv( hEnv=0:1 )
71      ---> Time elapsed - +1.129000E-003 seconds

72  SQLFreeEnv( )
73      <--- SQL_SUCCESS   Time elapsed - +2.870000E-004 seconds
```

JDBC trace file interpretation:

DB2 JDBC traces always begin with a header that lists important system information such as key environment variable settings, the JDK or JRE level, the DB2 JDBC driver level, and the DB2 build level. For example:

```
1   ========================================================
2   |    Trace beginning on 2002-1-28 7:21:0.19
3   ========================================================

4   System Properties:
5   ------------------
6   user.language = en
7   java.home = c:\Program Files\SQLLIB\java\jdk\bin\..
8   java.vendor.url.bug =
9   awt.toolkit = sun.awt.windows.WToolkit
10  file.encoding.pkg = sun.io
11  java.version = 1.1.8
12  file.separator = \
```

```
13  line.separator =
14  user.region = US
15  file.encoding = Cp1252
16  java.compiler = ibmjitc
17  java.vendor = IBM® Corporation
18  user.timezone = EST
19  user.name = db2user
20  os.arch = x86
21  java.fullversion = JDK 1.1.8 IBM build n118p-19991124 (JIT ibmjitc
                       V3.5-IBMJDK1.1-19991124)
22  os.name = Windows® NT
23  java.vendor.url = http://www.ibm.com/
24  user.dir = c:\Program Files\SQLLIB\samples\java
25  java.class.path =
         .:C:\Program Files\SQLLIB\lib;C:\Program Files\SQLLIB\java;
         C:\Program Files\SQLLIB\java\jdk\bin\
26  java.class.version = 45.3
27  os.version = 5.0
28  path.separator = ;
29  user.home = C:\home\db2user
30  ----------------------------------------
```

Note: Trace examples used in this section have line numbers added to the left hand side of the trace. These line numbers have been added to aid the discussion and will *not* appear in an actual DB2 JDBC trace.

Immediately following the trace header, one usually finds a number of trace entries related to initialization of the JDBC environment and database connection establishment. For example:

```
31  jdbc.app.DB2Driver -> DB2Driver() (2002-1-28 7:21:0.29)
32  | Loaded db2jdbc from java.library.path
33  jdbc.app.DB2Driver <- DB2Driver() [Time Elapsed = 0.01]

34  DB2Driver - connect(jdbc:db2:sample)

35  jdbc.app.DB2ConnectionTrace -> connect( sample, info, db2driver, 0, false )
                  (2002-1-28 7:21:0.59)
36  | 10: connectionHandle = 1
37  jdbc.app.DB2ConnectionTrace <- connect() [Time Elapsed = 0.16]

38  jdbc.app.DB2ConnectionTrace -> DB2Connection (2002-1-28 7:21:0.219)
39  | source = sample
40  | Connection handle = 1
41  jdbc.app.DB2ConnectionTrace <- DB2Connection
```

In the above trace example, a request to load the DB2 JDBC driver was made on line 31. This request returned successfully as reported on line 33.

The DB2 JDBC trace facility uses specific Java classes to capture the trace information. In the above trace example, one of those trace classes, DB2ConnectionTrace, has generated two trace entries numbered 35-37 and 38-41.

Line 35 shows the connect() method being invoked and the input parameters to that method call. Line 37 shows that the connect() method call has returned successfully while line 36 shows the output parameter of that call (Connection handle = 1).

Following the connection related entries, one usually finds statement related entries in the JDBC trace. For example:

```
42  jdbc.app.DB2ConnectionTrace -> createStatement() (2002-1-28 7:21:0.219)
43  | Connection handle = 1
44  | jdbc.app.DB2StatementTrace -> DB2Statement( con, 1003, 1007 )
```

```
          (2002-1-28 7:21:0.229)
45  | jdbc.app.DB2StatementTrace <- DB2Statement() [Time Elapsed = 0.0]
46  | jdbc.app.DB2StatementTrace -> DB2Statement (2002-1-28 7:21:0.229)
47  | | Statement handle = 1:1
48  | jdbc.app.DB2StatementTrace <- DB2Statement
49  jdbc.app.DB2ConnectionTrace <- createStatement - Time Elapsed = 0.01

50  jdbc.app.DB2StatementTrace -> executeQuery(SELECT * FROM EMPLOYEE WHERE
                                  empno = 000010) (2002-1-28 7:21:0.269)
51  | Statement handle = 1:1
52  | jdbc.app.DB2StatementTrace -> execute2( SELECT * FROM EMPLOYEE WHERE
                                    empno = 000010 ) (2002-1-28 7:21:0.269)
52  | | jdbc.DB2Exception -> DB2Exception() (2002-1-28 7:21:0.729)
53  | | | 10: SQLError = [IBM][CLI Driver][DB2/NT] SQL0401N  The data types of
              the operands for the operation "=" are not compatible.
              SQLSTATE=42818
54  | | |     SQLState = 42818
55  | | |     SQLNativeCode = -401
56  | | |     LineNumber = 0
57  | | |     SQLerrmc = =
58  | | jdbc.DB2Exception <- DB2Exception() [Time Elapsed = 0.0]
59  | jdbc.app.DB2StatementTrace <- executeQuery - Time Elapsed = 0.0
```

On line 42 and 43, the DB2ConnectionTrace class reported that the JDBC createStatement() method had been called with connection handle 1. Within that method, the internal method DB2Statement() was called as reported by another DB2 JDBC trace facility class, DB2StatementTrace. Notice that this internal method call appears 'nested' in the trace entry. Lines 47-49 show that the methods returned successfully and that statement handle 1:1 was allocated.

On line 50, an SQL query method call is made on statement 1:1, but the call generates an exception at line 52. The error message is reported on line 53 and contains the DB2 native error code that was generated (SQL0401N), the sqlstate that corresponds to that code (SQLSTATE=42818) and a brief description of the error. In this example, the error results because the EMPLOYEE.EMPNO column is defined as CHAR(6) and not an integer value as assumed in the query.

Related concepts:

- "CLI/ODBC/JDBC trace facility" on page 460

Related reference:

- "Miscellaneous variables" in the *Administration Guide: Performance*
- "Trace CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*
- "TraceComm CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*
- "TraceFileName CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*
- "TracePathName CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*
- "TracePIDList CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*
- "TraceRefreshInterval CLI/ODBC configuration keyword" in the *CLI Guide and Reference, Volume 1*

Chapter 21. Java 2 Platform Enterprise Edition

The sections that follow describe the Java 2 Platform Enterprise Edition (J2EE).

Java 2 Platform Enterprise Edition (J2EE) Overview

In today's global business environment, organizations need to extend their reach, lower their costs, and lower their response times by providing services that are easily accessible to their customers, employees, suppliers, and other business partners. These services need to have the following characteristics:

- Highly available, to meet the requirements of global business environment
- Secure, to protect the privacy of the users and the integrity of the enterprise
- Reliable and scalable, so that business transactions are accurately and promptly processed

In most cases, these services are provided with the help of multi-tier applications with each tier serving a specific purpose. The Java™ 2 Platform Enterprise Edition, reduces the cost and complexity of developing these multi-tier services, resulting in services that can be rapidly deployed and easily enhanced based on the requirements of the enterprise.

Java 2 Enterprise Edition achieves these benefits by defining a standard architecture that is delivered as the following elements:

- Java 2 Enterprise Edition Application Model, a standard application model for developing multi-tier, thin-client services
- Java 2 Enterprise Edition Platform, a standard platform for hosting Java 2 Enterprise Edition applications
- Java 2 Enterprise Edition Compatibility Test Suite for verifying that a Java 2 Enterprise Edition platform product complies with the Java 2 Enterprise Edition platform standard
- Java 2 Enterprise Edition Reference Implementation for demonstrating the capabilities of Java 2 Enterprise Edition, and for providing an operational definition of the Java 2 Enterprise Edition platform

Related concepts:

• "Java 2 Platform Enterprise Edition" on page 475

Java 2 Platform Enterprise Edition

The Java™ 2 Platform Enterprise Edition provides the runtime environment for hosting Java 2 Enterprise Edition applications. The runtime environment defines four application component types that a Java 2 Enterprise Edition product must support:

- Application clients are Java programming language programs that are typically GUI programs that execute on a desktop computer. Application clients have access to all of the facilities of the Java 2 Enterprise Edition middle tier.
- Applets are GUI components that typically execute in a web browser, but can execute in a variety of other applications or devices that support the applet programming model.

- Servlets, JavaServer Pages (JSPs), filters, and web event listeners typically execute in a web server and may respond to HTTP requests from web clients. Servlets, JSPs, and filters may be used to generate HTML pages that are an application's user interface. They may also be used to generate XML or other format data that is consumed by other application components. Servlets, pages created with the JSP technology, web filters, and web event listeners are referred to collectively in this specification as *web components*. Web applications are composed of web components and other data such as HTML pages.
- Enterprise JavaBeans™ (EJB) components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a Java 2 Enterprise Edition application.

The application components listed above can divided into three categories, based on how they can be deployed and managed:

- Components that are deployed, managed, and executed on a Java 2 Enterprise Edition server.
- Components that are deployed, managed on a Java 2 Enterprise Edition server, but are loaded to and executed on a client machine.
- Components whose deployment and management are not completely defined by this specification. Application clients can be under this category.

The runtime support for these components is provided by *containers*.

Related concepts:

- "Java 2 Platform Enterprise Edition Containers" on page 476
- "Enterprise Java Beans" on page 483

Java 2 Platform Enterprise Edition Containers

A container provides a federated view of the underlying Java™ 2 Platform Enterprise Edition APIs to the application components. A typical Java 2 Platform Enterprise Edition product will provide a container for each application component type; application client container, applet container, web container, and enterprise bean container. The container tools also understand the file formats for packaging the application components for deployment.

The specification requires that these containers provide a Java-compatible runtime environment, as defined by the Java 2 Platform Enterprise Edition, Standard Edition V1.3.1 specification J2SE. This specification defines a set of standard services that each Java 2 Enterprise Edition product must support. These standard services are:

- HTTP service
- HTTPS service
- Java transaction API
- Remote invocation method
- Java IDL
- JDBC API
- Java message service
- Java naming and directory interface
- JavaMail
- JavaBeans™ activation framework
- Java API for XML parsing
- Connector architecture
- Java authentication and authorization service

Related concepts:

- "Java Naming and Directory Interface (JNDI)" on page 477
- "Enterprise Java Beans" on page 483

Java 2 Platform Enterprise Edition Server

Underlying a Java™ 2 Platform Enterprise Edition container is the server of which the container is a part. A Java 2 Enterprise Edition Product Provider typically implements the Java 2 Platform Enterprise Edition server-side functionality using an existing transaction processing infrastructure in combination with J2SE technology. The Java 2 Platform Enterprise Edition client functionality is typically built on J2SE technology.

Note: The IBM® WebSphere® Application Server is a Java 2 Platform Enterprise Edition-compliant server.

Java 2 Enterprise Edition Database Requirements

The Java™ 2 Enterprise Edition platform requires a database, accessible through the JDBC API, for the storage of business data. The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.

Related concepts:

- Chapter 14, "Introduction to Java application support," on page 259

Java Naming and Directory Interface (JNDI)

JNDI enables Java™ platform-based applications to access multiple naming and directory services. It is a part of the Java Enterprise application programming interface (API) set. JNDI makes it possible for developers to create portable applications that are enabled for a number of different naming and directory services, including: file systems; directory services such as Lightweight Directory Access Protocol (LDAP), Novell Directory Services, and Network Information System (NIS); and distributed object systems such as the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Enterprise JavaBeans™ (EJB).

The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services and a service provider interface to attach a provider of a naming and directory service.

Java Transaction Management

Java™ 2 Enterprise Edition simplifies application programming for distributed transaction management. Java 2 Enterprise Edition includes support for distributed transactions through two specifications, Java Transaction API and Java Transaction Service (JTS). JTA is a high-level, implementation-independent, protocol-independent API that allows applications and application servers to access transactions. In addition, the JTA is always enabled.

The DB2 Universal JDBC Driver and the DB2® JDBC Type 2 Driver for Linux, UNIX® and Windows® implement the JTA and JTS specifications.

JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.

JTS specifies the implementation of a Transaction Manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using IIOP.

JTA and JTS allow application Java 2 Enterprise Edition servers to take the burden of transaction management off of the component developer. Developers can define the transactional properties of EJB technology based components during design or deployment using declarative statements in the deployment descriptor. The application server takes over the transaction management responsibilities.

In the DB2 and WebSphere® Application Server environment, WebSphere Application Server assumes the role of transaction manager, and DB2 acts as a resource manager. WebSphere Application Server implements JTS and part of JTA, and the JDBC drivers also implement part of JTA so that WebSphere Application Server and DB2 can provide coordinated distributed transactions.

It is not necessary to configure DB2 to be JTA-enabled in the WebSphere Application Server environment because the JDBC drivers automatically detect this environment.

The DB2 JDBC Type 2 Driver provides these two `DataSource` classes:
- `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`
- `COM.ibm.db2.jdbc.DB2XADataSource`

The DB2 Universal JDBC Driver provides these two `DataSource` classes:
- `com.ibm.db2.jcc.DB2ConnectionPoolDataSource`
- `com.ibm.db2.jcc.DB2XADataSource`

WebSphere Application Server provides pooled DB2 connections to databases. If the application will be involved in a distributed transaction, the `COM.ibm.db2.jdbc.DB2XADataSource` class should be used when defining DB2 data sources within the WebSphere Application Server.

For the detail information about how to configure the WebSphere Application Server with DB2, refer to WebSphere Application Server InfoCenter at:
`http://www-4.ibm.com/software/webservers/appserv/library.html`

Example of a distributed transaction that uses JTA methods

The best way to demonstrate distributed transactions is to contrast them with local transactions. With local transactions, a JDBC application makes changes to a database permanent and indicates the end of a unit of work in one of the following ways:

- By calling the `Connection.commit` or `Connection.rollback` methods after executing one or more SQL statements
- By calling the `Connection.setAutoCommit(true)` method at the beginning of the application to commit changes after every SQL statement

Figure 61 on page 479 outlines code that executes local transactions.

```
con1.setAutoCommit(false);  // Set autocommit off
// execute some SQL
...
con1.commit();              // Commit the transaction
// execute some more SQL
...
con1.rollback();            // Roll back the transaction
con1.setAutoCommit(true);   // Enable commit after every SQL statement
...
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

Figure 61. Example of a local transaction

In contrast, applications that participate in distributed transactions cannot call the `Connection.commit`, `Connection.rollback`, or `Connection.setAutoCommit(true)` methods within the distributed transaction. With distributed transactions, the `Connection.commit` or `Connection.rollback` methods do not indicate transaction boundaries. Instead, your applications let the application server manage transaction boundaries. Distributed transactions typically involve multiple connections to the same data source or different data sources, which can include data sources from different manufacturers.

Figure 62 demonstrates an application that uses distributed transactions. While the code in the example is running, the application server is also executing other EJBs that are part of this same distributed transaction. When all EJBs have called `utx.commit()`, the entire distributed transaction is committed by the application server. If any of the EJBs are unsuccessful, the application server rolls back all the work done by all EJBs that are associated with the distributed transaction.

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
...
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
...
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.

utx.commit();
...
```

Figure 62. Example of a distributed transaction under an application server

Figure 63 on page 480 illustrates a program that uses JTA methods to execute a distributed transaction. This program acts as the transaction manager and a transactional application. Two connections to two different data sources do SQL work under a single distributed transaction.

```
class XASample
{
  javax.sql.XADataSource xaDS1;
  javax.sql.XADataSource xaDS2;
  javax.sql.XAConnection xaconn1;
  javax.sql.XAConnection xaconn2;
  javax.transaction.xa.XAResource xares1;
  javax.transaction.xa.XAResource xares2;
  java.sql.Connection conn1;
  java.sql.Connection conn2;

  public static void main (String args []) throws java.sql.SQLException
  {
    XASample xat = new XASample();
    xat.runThis(args);
  }
  // As the transaction manager, this program supplies the global
  // transaction ID and the branch qualifier. The global
  // transaction ID and the branch qualifier must not be
  // equal to each other, and the combination must be unique for
  // this transaction manager.
  public void runThis(String[] args)
  {
    byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
    byte[] bqual = new byte[] { 0x00, 0x22, 0x00 };
    int rc1 = 0;
    int rc2 = 0;

    try
    {

      javax.naming.InitialContext context = new javax.naming.InitialContext();
      /*
        * Note that javax.sql.XADataSource is used instead of a specific
        * driver implementation such as com.ibm.db2.jcc.DB2XADataSource,
        * which can be used only if this is a DB2 connection.
        */
      xaDS1 = (javax.sql.XADataSource)context.lookup("checkingAccounts");
      xaDS2 = (javax.sql.XADataSource)context.lookup("savingsAccounts");

      // The XADatasource contains the user ID and password.
      // Get the XAConnection object from each XADataSource
      xaconn1 = xaDS1.getXAConnection();
      xaconn2 = xaDS2.getXAConnection();

      // Get the java.sql.Connection object from each XAConnection
      conn1 = xaconn1.getConnection();
      conn2 = xaconn2.getConnection();

      // Get the XAResource object from each XAConnection
      xares1 = xaconn1.getXAResource();
      xares2 = xaconn2.getXAResource();
```

Figure 63. Example of a distributed transaction that uses the JTA (Part 1 of 4)

```
      // Create the Xid object for this distributed transaction.
      // This example uses the com.ibm.db2.jcc.DB2Xid implementation
      // of the Xid interface. This Xid can be used with any JDBC driver
      // that supports JTA.
      javax.transaction.xa.Xid xid1 =
                              new com.ibm.db2.jcc.DB2Xid(100, gtrid, bqual);

      // Start the distributed transaction on the two connections.
      // The two connections do NOT need to be started and ended together.
      // They might be done in different threads, along with their SQL operations.
      xares1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
      xares2.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
...
      // Do the SQL operations on connection 1.
      // Do the SQL operations on connection 2.
...
      // Now end the distributed transaction on the two connections.
      xares1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
      xares2.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);

      // If connection 2 work had been done in another thread,
      // a thread.join() call would be needed here to wait until the
      // connection 2 work is done.

      try
      { // Now prepare both branches of the distributed transaction.
        // Both branches must prepare successfully before changes
        // can be committed.
        // If the distributed transaction fails, an XAException is thrown.
        rc1 = xares1.prepare(xid1);
        if(rc1 == javax.transaction.xa.XAResource.XA_OK)
        { // Prepare was successful. Prepare the second connection.
          rc2 = xares2.prepare(xid1);
          if(rc2 == javax.transaction.xa.XAResource.XA_OK)
          { // Both connections prepared successfully and neither was read-only.
            xares1.commit(xid1, false);
            xares2.commit(xid1, false);
          }
          else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
          { // The second connection is read-only, so just commit the
            // first connection.
            xares1.commit(xid1, false);
          }
        }
        else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
        { // SQL for the first connection is read-only (such as a SELECT).
          // The prepare committed it. Prepare the second connection.
          rc2 = xares2.prepare(xid1);
          if(rc2 == javax.transaction.xa.XAResource.XA_OK)
          { // The first connection is read-only but the second is not.
            // Commit the second connection.
            xares2.commit(xid1, false);
          }
          else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
          { // Both connections are read-only, and both already committed,
            // so there is nothing more to do.
          }
        }
      }
```

Figure 63. Example of a distributed transaction that uses the JTA (Part 2 of 4)

```
  catch (javax.transaction.xa.XAException xae)
  { // Distributed transaction failed, so roll it back.
    // Report XAException on prepare/commit.
    System.out.println("Distributed transaction prepare/commit failed. " +
                       "Rolling it back.");
    System.out.println("XAException error code = " + xae.errorCode);
    System.out.println("XAException message = " + xae.getMessage());
    xae.printStackTrace();
    try
    {
      xares1.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae1)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares1 failed");
      System.out.println("XAException error code = " + xae1.errorCode);
      System.out.println("XAException message = " + xae1.getMessage());
    }
    try
    {
      xares2.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae2)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares2 failed");
      System.out.println("XAException error code = " + xae2.errorCode);
      System.out.println("XAException message = " + xae2.getMessage());
    }
  }

  try
  {
    conn1.close();
    xaconn1.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 1: " + e.toString());
    e.printStackTrace();
  }
  try
  {
    conn2.close();
    xaconn2.close();
  }
  catch (Exception e)
  {
    System.out.println("Failed to close connection 2: " + e.toString());
    e.printStackTrace();
  }
}
```

Figure 63. Example of a distributed transaction that uses the JTA (Part 3 of 4)

```
    catch (java.sql.SQLException sqe)
    {
      System.out.println("SQLException caught: " + sqe.getMessage());
      sqe.printStackTrace();
    }
    catch (javax.transaction.xa.XAException xae)
    {
      System.out.println("XA error is " + xae.getMessage());
      xae.printStackTrace();
    }
    catch (javax.naming.NamingException nme)
    {
      System.out.println(" Naming Exception: " + nme.getMessage());
    }
  }
}
```

Figure 63. Example of a distributed transaction that uses the JTA (Part 4 of 4)

Recommendation: For better performance, complete a distributed transaction before you start another distributed or local transaction.

Related concepts:

- "Java Transaction Management" on page 477

Enterprise Java Beans

The Enterprise Java™ beans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications that are written using the Enterprise Java beans architecture can be written once, and then deployed on any server platform that supports the Enterprise Java beans specification. Java 2 Enterprise Edition applications implement server-side business components using Enterprise Java beans (EJBs) that include session beans and entity beans.

Session beans represent business services and are not shared between users. Entity beans are multi-user, distributed transactional objects that represent persistent data. The transactional boundaries of a EJB application can be set by specifying either container-managed or bean-managed transactions.

The sample program `AccessEmployee.ear` uses Enterprise Java beans to implement a Java 2 Enterprise Edition application to access a DB2® database. You can find this sample in the `SQLLIB/samples/websphere` directory.

The EJB sample application provides two business services. One service allows the user to access information about an employee (which is stored in the EMPLOYEE table of the **sample** database) through that employee's employee number. The other service allows the user to retrieve a list of the employee numbers, so that the user can obtain an employee number to use for querying employee data.

The following sample uses EJBs to implement a Java 2 Enterprise Edition application to access a DB2 database. The sample utilizes the Model-View-Controller (MVC) architecture, which is a commonly-used GUI architecture. The JSP is used to implement the view (the presentation component). A servlet acts as the controller in the sample. It controls the workflow and delegates the user's request to the model, which is implemented using EJBs. The model component of the sample consists of two EJBs, one session bean and one entity bean. The

container-managed persistence (CMP) bean, Employee, represents the distributed transactional objects that represent the persistent data in the EMPLOYEE table of the sample database. The term container-managed persistence means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). The session bean, AccessEmployee, acts as the Façade of the entity bean and provides provide a uniform client access strategy. This Façade design reduces the network traffic between the EJB client and the entity bean and is more efficient in distributed transactions than if the EJB client accesses the entity bean directly. Access to the DB2 database can be provided from the session bean or entity bean. The two services of the sample application demonstrate both approaches to accessing the DB2 database. In the first service, the entity bean is used:

```
//=================================================
// This method returns an employee's information by
// interacting with the entity bean located by the
// provided employee number
public EmployeeInfo getEmployeeInfo(String empNo)
throws java.rmi.RemoteException
}
Employee employee = null;
try
}
employee = employeeHome.findByPrimaryKey(new EmployeeKey(empNo));
EmployeeInfo empInfo = new EmployeeInfo(empNo);
//set the employee's information to the dependent value object
empInfo.setEmpno(employee.getEmpno());
empInfo.setFirstName (employee.getFirstName());
empInfo.setMidInit(employee.getMidInit());
empInfo.setLastName(employee.getLastName());
empInfo.setWorkDept(employee.getWorkDept());
empInfo.setPhoneNo(employee.getPhoneNo());
empInfo.setHireDate(employee.getHireDate());
empInfo.setJob(employee.getJob());
empInfo.setEdLevel(employee.getEdLevel());
empInfo.setSex(employee.getSex());
empInfo.setBirthDate(employee.getBirthDate());
empInfo.setSalary(employee.getSalary());
empInfo.setBonus(employee.getBonus());
empInfo.setComm(employee.getComm());
return empInfo;
}
catch (java.rmi.RemoteException rex)
{
......
```

In the second service, which displays employee numbers, the session bean, AccessEmployee, directly accesses the DB2 sample database.

```
/=============================================
* Get the employee number list.
* @return Collection
*/
public Collection getEmpNoList()
{
ResultSet rs = null;
PreparedStatement ps = null;
Vector list = new Vector();
DataSource ds = null;
Connection con = null;
try
{
ds = getDataSource();
```

```
con = ds.getConnection();
String schema = getEnvProps(DBschema);
String query = "Select EMPNO from " + schema + ".EMPLOYEE";
ps = con.prepareStatement(query);
ps.executeQuery();
rs = ps.getResultSet();
EmployeeKey pk;
while (rs.next())
{
pk = new EmployeeKey();
pk.employeeId = rs.getString(1);
list.addElement(pk.employeeId);
}
rs.close();
return list;
```

Related reference:

- "Java WebSphere samples" in the *Application Development Guide: Building and Running Applications*

Part 5. Other Programming Interfaces

Chapter 22. Programming in Perl

Programming Considerations for Perl

Perl is a popular programming language that is freely available for many operating systems. Using the DBD::DB2 driver available from http://www.ibm.com/software/data/db2/perl with the Perl Database Interface (DBI) Module available from http://www.perl.com, you can create DB2® applications using Perl.

Because Perl is an interpreted language and the Perl DBI Module uses dynamic SQL, Perl is an ideal language for quickly creating and revising prototypes of DB2 applications. The Perl DBI Module uses an interface that is quite similar to the CLI and JDBC interfaces, which makes it easy for you to port your Perl prototypes to CLI and JDBC.

Most database vendors provide a database driver for the Perl DBI Module, which means that you can also use Perl to create applications that access data from many different database servers. For example, you can write a Perl DB2 application that connects to an Oracle database using the DBD::Oracle database driver, fetch data from the Oracle database, and insert the data into a DB2 database using the DBD::DB2 database driver.

Perl Restrictions

The Perl DBI module supports only dynamic SQL. When you need to execute a statement multiple times, you can improve the performance of your Perl DB2® applications by issuing a **prepare** call to prepare the statement.

For current information on the restrictions of the version of the DBD::DB2 driver that you install on your workstation, refer to the CAVEATS file in the DBD::DB2 driver package.

Multiple-Thread Database Access in Perl

Perl does not support multiple-thread database access.

Database Connections in Perl

To enable Perl to load the DBI module, you must include the following line in your DB2® application:

```
use DBI;
```

The DBI module automatically loads the DBD::DB2 driver when you create a *database handle* using the **DBI->connect** statement with the following syntax:

```
my $dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password);
```

where:

$dbhandle
represents the database handle returned by the connect statement

dbalias
represents a DB2 alias cataloged in your DB2 database directory

$userID
represents the user ID used to connect to the database

$password
represents the password for the user ID used to connect to the database

Fetching Results in Perl

Because the Perl DBI Module only supports dynamic SQL, you do not use host variables in your Perl DB2 applications.

Procedure:

To return results from an SQL query, perform the following steps:

1. Create a database handle by connecting to the database with the `DBI->connect` statement.
2. Create a statement handle from the database handle. For example, you can call **prepare** with an SQL statement as a string argument to return statement handle *$sth* from the database handle, as demonstrated in the following Perl statement:

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee '
   );
```

3. Execute the SQL statement by calling **execute** on the statement handle. A successful call to **execute** associates a result set with the statement handle. For example, you can execute the statement prepared in the previous example using the following Perl statement:

```
#Note: $rc represents the return code for the execute call
my $rc = $sth->execute();
```

4. Fetch a row from the result set associated with the statement handle with a call to **fetchrow()**. The Perl DBI returns a row as an array with one value per column. For example, you can return all of the rows from the statement handle in the previous example using the following Perl statement:

```
while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme $lastname\n";
}
```

Related concepts:

- "Database Connections in Perl" on page 489

Parameter Markers in Perl

To enable you to execute a prepared statement using different input values for specified fields, the Perl DBI module enables you to prepare and execute a statement using parameter markers. To include a parameter marker in an SQL statement, use the question mark (?) character.

The following Perl code creates a statement handle that accepts a parameter marker for the WHERE clause of a SELECT statement. The code then executes the statement twice using the input values 25000 and 35000 to replace the parameter marker.

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee
      WHERE salary > ?'
   );

my $rc = $sth->execute(25000);

⋮
my $rc = $sth->execute(35000);
```

SQLSTATE and SQLCODE Variables in Perl

To return the SQLSTATE associated with a Perl DBI database handle or statement handle, call the **state** method. For example, to return the SQLSTATE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlstate = $dbhandle->state;
```

To return the SQLCODE associated with a Perl DBI database handle or statement handle, call the **err** method. To return the message for an SQLCODE associated with a Perl DBI database handle or statement handle, call the **errstr** method. For example, to return the SQLCODE associated with the database handle $dbhandle, include the following Perl statement in your application:

```
my $sqlcode = $dbhandle->err;
```

Example of a Perl Program

Following is an example of an application written in Perl:

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]\n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname\n";
}

# check for problems which may have terminated the fetch early
```

```
warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;
```

Chapter 23. Programming in REXX

Programming Considerations for REXX

Special host-language programming considerations are discussed in the following sections. Included is information on embedding SQL statements, language restrictions, and supported data types for host variables.

Note: REXX support stabilized in DB2 Version 5, and no enhancements for REXX support are planned for the future. For example, REXX cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than REXX.

Because REXX is an interpreted language, no precompiler, compiler, or linker is used. Instead, three DB2 APIs are used to create DB2 applications in REXX. Use these APIs to access different elements of DB2.

SQLEXEC
Supports the SQL language.

SQLDBS
Supports command-like versions of DB2 APIs.

SQLDB2
Supports a REXX specific interface to the command-line processor. See the description of the API syntax for REXX for details and restrictions on how this interface can be used.

Related concepts:

- "API Syntax for REXX" on page 505

Language Restrictions for REXX

The sections that follow describe the language restrictions for REXX.

Language Restrictions for REXX

It is possible that tokens within statements or commands that are passed to the SQLEXEC, SQLDBS, and SQLDB2 routines could correspond to REXX variables. In this case, the REXX interpreter substitutes the variable's value before calling SQLEXEC, SQLDBS, or SQLDB2.

To avoid this situation, enclose statement strings in quotation marks (' ' or " "). If you do not use quotation marks, any conflicting variable names are resolved by the REXX interpreter, instead of being passed to the SQLEXEC, SQLDBS or SQLDB2 routines.

Compound SQL is not supported in REXX/SQL.

REXX/SQL stored procedures are supported on Windows® operating systems, but not on AIX®.

Related tasks:

- "Registering SQLEXEC, SQLDBS and SQLDB2 in REXX" on page 494

Registering SQLEXEC, SQLDBS and SQLDB2 in REXX

Before using any of the DB2 APIs or issuing SQL statements in an application, you must register the SQLDBS, SQLDB2 and SQLEXEC routines. This notifies the REXX interpreter of the REXX/SQL entry points. The method you use for registering varies slightly between Windows-based and AIX platforms.

Procedure:

Use the following examples for correct syntax for registering each routine:

Sample registration on Windows-based platforms

```
/* ------------ Register SQLDBS with REXX  -------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy \= 0 then
    do
      say 'SQLDBS was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ------------ Register SQLDB2 with REXX  -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy \= 0 then
    do
      say 'SQLDB2 was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ----------------- Register SQLEXEC with REXX  --------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy \= 0 then
    do
      say 'SQLEXEC was not successfully added to the REXX environment'
      signal rxx_exit
    end
```

Sample registration on AIX

```
 /* ------------ Register SQLDBS, SQLDB2 and SQLEXEC with REXX --------*/
rcy = SysAddFuncPkg("db2rexx")
If rcy \= 0 then
  do
    say 'db2rexx was not successfully added to the REXX environment'
    signal rxx_exit
  end
```

On Windows-based platforms, the RxFuncAdd commands need to be executed only once for all sessions.

On AIX, the SysAddFuncPkg should be executed in every REXX/SQL application.

Details on the RXfuncadd and SysAddFuncPkg APIs are available in the REXX documentation for Windows-based platforms and AIX, respectively.

Multiple-Thread Database Access in REXX

REXX does not support multiple-thread database access.

Japanese or Traditional Chinese EUC Considerations for REXX

REXX applications are not supported under Japanese or Traditional Chinese EUC environments.

Embedded SQL in REXX Applications

REXX applications use APIs that enable them to use most of the features provided by database manager APIs and SQL. Unlike applications written in a compiled language, REXX applications are not precompiled. Instead, a dynamic SQL handler processes all SQL statements. By combining REXX with these callable APIs, you have access to most of the database manager capabilities. Although REXX does not directly support some APIs using embedded SQL, they can be accessed using the DB2® command line processor from within the REXX application.

As REXX is an interpretive language, you may find it is easier to develop and debug your application prototypes in REXX as compared to compiled host languages. Although DB2 applications coded in REXX do not provide the performance of DB2 applications that use compiled languages, they do provide the ability to create DB2 applications without precompiling, compiling, linking, or using additional software.

Use the SQLEXEC routine to process all SQL statements. The character string arguments for the SQLEXEC routine are made up of the following elements:

- SQL keywords
- Pre-declared identifiers
- Statement host variables

Make each request by passing a valid SQL statement to the SQLEXEC routine. Use the following syntax:

```
CALL SQLEXEC 'statement'
```

SQL statements can be continued onto more than one line. Each part of the statement should be enclosed in single quotation marks, and a comma must delimit additional statement text as follows:

```
CALL SQLEXEC 'SQL text',
             'additional text',
                  .
                  .
                  .
             'final text'
```

The following is an example of embedding an SQL statement in REXX:

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

In this example, the SQLCODE field of the SQLCA structure is checked to determine whether the update was successful.

The following rules apply to embedded SQL statements:

- The following SQL statements can be passed directly to the SQLEXEC routine:
  - CALL
  - CLOSE
  - COMMIT
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - DECLARE
  - DESCRIBE
  - DISCONNECT
  - EXECUTE
  - EXECUTE IMMEDIATE
  - FETCH
  - FREE LOCATOR
  - OPEN
  - PREPARE
  - RELEASE
  - ROLLBACK
  - SET CONNECTION

  Other SQL statements must be processed dynamically using the EXECUTE IMMEDIATE, or PREPARE and EXECUTE statements in conjunction with the SQLEXEC routine.
- You cannot use host variables in the CONNECT and SET CONNECTION statements in REXX.
- Cursor names and statement names are predefined as follows:

  c1 to c100
  Cursor names, which range from *c1* to *c50* for cursors declared without the WITH HOLD option, and *c51* to *c100* for cursors declared using the WITH HOLD option.

  The cursor name identifier is used for DECLARE, OPEN, FETCH, and CLOSE statements. It identifies the cursor used in the SQL request.

  s1 to s100
  Statement names, which range from *s1* to *s100*.

The statement name identifier is used with the DECLARE, DESCRIBE, PREPARE, and EXECUTE statements.

The pre-declared identifiers must be used for cursor and statement names. Other names are not allowed.

- When declaring cursors, the cursor name and the statement name should correspond in the DECLARE statement. For example, if *c1* is used as a cursor name, *s1* must be used for the statement name.
- Do not use comments within an SQL statement.

Host Variables in REXX

The sections that follow describe how to declare and use host variables in REXX programs.

Host Variables in REXX

Host variables are REXX language variables that are referenced within SQL statements. They allow an application to pass input data to DB2 and receive output data from DB2. REXX applications do not need to declare host variables, except for LOB locators and LOB file reference variables. Host variable data types and sizes are determined at run time when the variables are referenced. The sections that follow describe the rules to follow when naming and using host variables.

Related concepts:

- "Host Variable Names in REXX" on page 497
- "Host Variable References in REXX" on page 497
- "Indicator Variables in REXX" on page 498
- "LOB Host Variables in REXX" on page 500
- "LOB Host Variable Clearing in REXX" on page 502

Related reference:

- "Predefined REXX Variables" on page 498
- "Syntax for LOB Locator Declarations in REXX" on page 500
- "Syntax for LOB File Reference Declarations in REXX" on page 501
- "Supported SQL Data Types in REXX" on page 502

Host Variable Names in REXX

Any properly named REXX variable can be used as a host variable. A variable name can be up to 64 characters long. Do not end the name with a period. A host variable name can consist of alphabetic characters, numerics, and the characters @, _, !, ., ?, and $.

Host Variable References in REXX

The REXX interpreter examines every string without quotation marks in a procedure. If the string represents a variable in the current REXX variable pool, REXX replaces the string with the current value. The following is an example of how you can reference a host variable in REXX:

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

To ensure that a character string is not converted to a numeric data type, enclose the string with single quotation marks as in the following example:

```
VAR = '100'
```

REXX sets the variable *VAR* to the 3-byte character string `100`. If single quotation marks are to be included as part of the string, follow this example:

```
VAR = "'100'"
```

When inserting numeric data into a CHARACTER field, the REXX interpreter treats numeric data as integer data, thus you must concatenate numeric strings explicitly and surround them with single quotation marks.

Indicator Variables in REXX

An indicator variable data type in REXX is a number without a decimal point. Following is an example of an indicator variable in REXX using the INDICATOR keyword.

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

In the above example, `cmind` is examined for a negative value. If it is not negative, the application can use the returned value of `cm`. If it is negative, the fetched value is NULL and `cm` should not be used. The database manager does not change the value of the host variable in this case.

Predefined REXX Variables

SQLEXEC, SQLDBS, and SQLDB2 set predefined REXX variables as a result of certain operations. These variables are:

RESULT

Each operation sets this return code. Possible values are:

n Where *n* is a positive value indicating the number of bytes in a formatted message. The GET ERROR MESSAGE API alone returns this value.

0 The API was executed. The REXX variable SQLCA contains the completion status of the API. If SQLCA.SQLCODE is not zero, SQLMSG contains the text message associated with that value.

–1 There is not enough memory available to complete the API. The requested message was not returned.

–2 SQLCA.SQLCODE is set to 0. No message was returned.

–3 SQLCA.SQLCODE contained an invalid SQLCODE. No message was returned.

–6 The SQLCA REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.

–7 The SQLMSG REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.

–8 The SQLCA.SQLCODE REXX variable could not be fetched from the REXX variable pool.

–9 The SQLCA.SQLCODE REXX variable was truncated during the fetch. The maximum length for this variable is 5 bytes.

–10 The SQLCA.SQLCODE REXX variable could not be converted from ASCII to a valid long integer.

–11 The SQLCA.SQLERRML REXX variable could not be fetched from the REXX variable pool.

–12 The SQLCA.SQLERRML REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.

–13 The SQLCA.SQLERRML REXX variable could not be converted from ASCII to a valid short integer.

–14 The SQLCA.SQLERRMC REXX variable could not be fetched from the REXX variable pool.

–15 The SQLCA.SQLERRMC REXX variable was truncated during the fetch. The maximum length for this variable is 70 bytes.

–16 The REXX variable specified for the error text could not be set.

–17 The SQLCA.SQLSTATE REXX variable could not be fetched from the REXX variable pool.

–18 The SQLCA.SQLSTATE REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.

Note: The values –8 through –18 are returned only by the GET ERROR MESSAGE API.

SQLMSG
If SQLCA.SQLCODE is not 0, this variable contains the text message associated with the error code.

SQLISL
The isolation level. Possible values are:

RR Repeatable read.
RS Read stability.
CS Cursor stability. This is the default.
UR Uncommitted read.
NC No commit. (NC is only supported by some host, AS/400, or iSeries servers.)

SQLCA
The SQLCA structure updated after SQL statements are processed and DB2 APIs are called.

SQLRODA
The input/output SQLDA structure for stored procedures invoked using the CALL statement. It is also the output SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API.

SQLRIDA
The input SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API.

SQLRDAT
An SQLCHAR structure for server procedures invoked using the Database Application Remote Interface (DARI) API.

Related reference:

- "SQLCA" in the *Administrative API Reference*
- "SQLCHAR" in the *Administrative API Reference*
- "SQLDA" in the *Administrative API Reference*

LOB Host Variables in REXX

When you fetch a LOB column into a REXX host variable, it will be stored as a simple (that is, uncounted) string. This is handled in the same manner as all character-based SQL types (such as CHAR, VARCHAR, GRAPHIC, LONG, and so on). On input, if the size of the contents of your host variable is larger than 32K, or if it meets other criteria set out below, it will be assigned the appropriate LOB type.

In REXX SQL, LOB types are determined from the string content of your host variable as follows:

Host variable string content	Resulting LOB type
:hv1='ordinary quoted string longer than 32K ...'	CLOB
:hv2="'string with embedded delimiting quotation marks ", "longer than 32K...'"	CLOB
:hv3="G'DBCS string with embedded delimiting single ", "quotation marks, beginning with G, longer than 32K...'"	DBCLOB
:hv4="BIN'string with embedded delimiting single ", "quotation marks, beginning with BIN, any length...'"	BLOB

Syntax for LOB Locator Declarations in REXX

The following shows the syntax for declaring LOB locator host variables in REXX:

Syntax for LOB Locator Host Variables in REXX

You must declare LOB locator host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as locators for the remainder of the program. Locator values are stored in REXX variables in an internal format.

Example:

```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

Data represented by LOB locators returned from the engine can be freed in REXX/SQL using the FREE LOCATOR statement which has the following format:

Syntax for FREE LOCATOR Statement

Example:

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

Syntax for LOB File Reference Declarations in REXX

You must declare LOB file reference host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as LOB file references for the remainder of the program.

The following shows the syntax for declaring LOB file reference host variables in REXX:

REXX File Reference Declarations

Example:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

File reference variables in REXX contain three fields. For the above example they are:

hv3.FILE_OPTIONS.
: Set by the application to indicate how the file will be used.

hv3.DATA_LENGTH.
: Set by DB2 to indicate the size of the file.

hv3.NAME.
: Set by the application to the name of the LOB file.

For FILE_OPTIONS, the application sets the following keywords:

Keyword (Integer Value)	Meaning
READ (2)	File is to be used for input. This is a regular file that can be opened, read and closed. The length of the data in the file (in bytes) is computed (by the application requestor code) upon opening the file.
CREATE (8)	On output, create a new file. If the file already exists, it is an error. The length (in bytes) of the file is returned in the DATA_LENGTH field of the file reference variable structure.
OVERWRITE (16)	On output, the existing file is overwritten if it exists, otherwise a new file is created. The length (in bytes) of the file is returned in the DATA_LENGTH field of the file reference variable structure.

APPEND (32) The output is appended to the file if it exists, otherwise a new file is created. The length (in bytes) of the data that was added to the file (not the total file length) is returned in the DATA_LENGTH field of the file reference variable structure.

Note: A file reference host variable is a compound variable in REXX, thus you must set values for the NAME, NAME_LENGTH and FILE_OPTIONS fields in addition to declaring them.

LOB Host Variable Clearing in REXX

On Windows®-based platforms it may be necessary to explicitly clear REXX SQL LOB locator and file reference host variable declarations as they remain in effect after your application program ends. This occurs because the application process does not exit until the session in which it is run is closed. If REXX SQL LOB declarations are not cleared, they may interfere with other applications that are running in the same session after a LOB application has been executed.

The syntax to clear the declaration is:

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

You should code this statement at the end of LOB applications. Note that you can code it anywhere as a precautionary measure to clear declarations which might have been left by previous applications (for example, at the beginning of a REXX SQL application).

Cursors in REXX

When a cursor is declared in REXX, the cursor is associated with a query. The query is associated with a statement name assigned in the PREPARE statement. Any referenced host variables are represented by parameter markers. The following example shows a DECLARE statement associated with a dynamic SELECT statement:

```
prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"
CALL SQLEXEC 'PREPARE S1 FROM :prep_string';
CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';
CALL SQLEXEC 'OPEN C1 USING :schema_name';
```

Related reference:

- "Supported SQL Data Types in REXX" on page 502

Supported SQL Data Types in REXX

Certain predefined REXX data types correspond to DB2 column types. The following table shows how SQLEXEC and SQLDBS interpret REXX variables in order to convert their contents to DB2 data types.

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

Table 80. SQL Column Types Mapped to REXX Declarations

SQL Column Type[1]	REXX Data Type	SQL Column Type Description
SMALLINT (500 or 501)	A number without a decimal point ranging from -32 768 to 32 767	16-bit signed integer

Table 80. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type[1]	REXX Data Type	SQL Column Type Description
INTEGER (496 or 497)	A number without a decimal point ranging from -2 147 483 648 to 2 147 483 647	32-bit signed integer
REAL[2] (480 or 481)	A number in scientific notation ranging from -3.40282346 x 10^{38} to 3.40282346 x 10^{38}	Single-precision floating point
DOUBLE[3] (480 or 481)	A number in scientific notation ranging from -1.79769313 x 10^{308} to 1.79769313 x 10^{308}	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	A number with a decimal point	Packed decimal
CHAR(*n*) (452 or 453)	A string with a leading and trailing quotation mark (’), which has length *n* after removing the two quotation marks A string of length *n* with any non-numeric characters, other than leading and trailing blanks or the E in scientific notation	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	Equivalent to CHAR(*n*)	Variable-length character string of length *n*, where *n* ranges from 1 to 4000
LONG VARCHAR (456 or 457)	Equivalent to CHAR(*n*)	Variable-length character string of length *n*, where *n* ranges from 1 to 32 700
CLOB(*n*) (408 or 409)	Equivalent to CHAR(*n*)	Large object variable-length character string of length *n*, where *n* ranges from 1 to 2 147 483 647
CLOB locator variable[4] (964 or 965)	DECLARE *:var_name* LANGUAGE TYPE CLOB LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable[4] (920 or 921)	DECLARE *:var_name* LANGUAGE TYPE CLOB FILE	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	A string with a leading and trailing apostrophe, preceded by BIN, containing *n* characters after removing the preceding BIN and the two apostrophes.	Large object variable-length binary string of length *n*, where *n* ranges from 1 to 2 147 483 647
BLOB locator variable[4] (960 or 961)	DECLARE *:var_name* LANGUAGE TYPE BLOB LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable[4] (916 or 917)	DECLARE *:var_name* LANGUAGE TYPE BLOB FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	Equivalent to CHAR(*10*)	10-byte character string
TIME (388 or 389)	Equivalent to CHAR(*8*)	8-byte character string
TIMESTAMP (392 or 393)	Equivalent to CHAR(*26*)	26-byte character string
Note: The following data types are only available in the DBCS environment.		
GRAPHIC(*n*) (468 or 469)	A string with a leading and trailing apostrophe preceded by a G or N, containing *n* DBCS characters after removing the preceding character and the two apostrophes	Fixed-length graphic string of length *n*, where *n* is from 1 to 127
VARGRAPHIC(*n*) (464 or 465)	Equivalent to GRAPHIC(*n*)	Variable-length graphic string of length *n*, where *n* ranges from 1 to 2 000
LONG VARGRAPHIC (472 or 473)	Equivalent to GRAPHIC(*n*)	Long variable-length graphic string of length *n*, where *n* ranges from 1 to 16 350
DBCLOB(*n*) (412 or 413)	Equivalent to GRAPHIC(*n*)	Large object variable-length graphic string of length *n*, where *n* ranges from 1 to 1 073 741 823

Table 80. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type[1]	REXX Data Type	SQL Column Type Description
DBCLOB locator variable[4] (968 or 969)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB LOCATOR	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[4] (924 or 925)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB FILE	Descriptor for file containing DBCLOB data

Notes:

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string.
2. FLOAT(n) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(n) where $24 < n < 54$ is
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

Related concepts:

- "Cursors in REXX" on page 502

Execution Requirements for REXX

The sections that follow describe the execution requirements for REXX applications.

Building and Running REXX Applications

REXX applications are not precompiled, compiled, or linked. The instructions below describe how to build and run REXX applications on Windows operating systems, and on the AIX operating system.

Restrictions:

On Windows-based platforms, your application file must have a .CMD extension. After creation, you can run your application directly from the operating system command prompt. On AIX, your application file can have any extension.

Procedure:

Build and run your REXX applications as follows:

- On Windows operating systems, your application file can have any name. After creation, you can run your application from the operating system command prompt by invoking the REXX interpreter as follows:

  ```
  REXX file_name
  ```

- On AIX, you can run your application using either of the following two methods:
  - At the shell command prompt, type `rexx name` where `name` is the name of your REXX program.
  - If the first line of your REXX program contains a "magic number" (#!) and identifies the directory where the REXX/6000 interpreter resides, you can run your REXX program by typing its name at the shell command prompt. For example, if the REXX/6000 interpreter file is in the /usr/bin directory, include the following as the very first line of your REXX program:

```
#! /usr/bin/rexx
```

Then, make the program executable by typing the following command at the shell command prompt:

```
chmod +x name
```

Run your REXX program by typing its file name at the shell command prompt.

Note: On AIX, you should set the LIBPATH environment variable to include the directory where the REXX SQL library, `db2rexx` is located. For example:

```
export LIBPATH=/lib:/usr/lib:/usr/lpp/db2_08_01/lib
```

Bind Files for REXX

Five bind files are provided to support REXX applications. The names of these files are included in the DB2UBIND.LST file. Each bind file was precompiled using a different isolation level; therefore, there are five different packages stored in the database.

The five bind files are:

DB2ARXCS.BND
: Supports the cursor stability isolation level.

DB2ARXRR.BND
: Supports the repeatable read isolation leve.l

DB2ARXUR.BND
: Supports the uncommitted read isolation level.

DB2ARXRS.BND
: Supports the read stability isolation level.

DB2ARXNC.BND
: Supports the no commit isolation level. This isolation level is used when working with some host, AS/400, or iSeries database servers. On other databases, it behaves like the uncommitted read isolation level.

Note: In some cases, it may be necessary to explicitly bind these files to the database.

When you use the SQLEXEC routine, the package created with cursor stability is used as a default. If you require one of the other isolation levels, you can change isolation levels with the SQLDBS CHANGE SQL ISOLATION LEVEL API, before connecting to the database. This will cause subsequent calls to the SQLEXEC routine to be associated with the specified isolation level.

Windows-based REXX applications cannot assume that the default isolation level is in effect unless they know that no other REXX programs in the session have changed the setting. Before connecting to a database, a REXX application should explicitly set the isolation level.

API Syntax for REXX

Use the SQLDBS routine to call DB2 APIs with the following syntax:

```
CALL SQLDBS 'command string'
```

If a DB2® API you want to use cannot be called using the SQLDBS routine, you may still call the API by calling the DB2 command line processor (CLP) from within the REXX application. However, because the DB2 CLP directs output either

to the standard output device or to a specified file, your REXX application cannot directly access the output from the called DB2 API, nor can it easily make a determination as to whether the called API is successful or not. The SQLDB2 API provides an interface to the DB2 CLP that provides direct feedback to your REXX application on the success or failure of each called API by setting the compound REXX variable, SQLCA, after each call.

You can use the SQLDB2 routine to call DB2 APIs using the following syntax:

```
CALL SQLDB2 'command string'
```

where `'command string'` is a string that can be processed by the command-line processor (CLP).

Calling a DB2 API using SQLDB2 is equivalent to calling the CLP directly, except for the following:

- The call to the CLP executable is replaced by the call to SQLDB2 (all other CLP options and parameters are specified the same way).
- The REXX compound variable SQLCA is set after calling the SQLDB2 but is not set after calling the CLP executable.
- The default display output of the CLP is set to off when you call SQLDB2, whereas the display is set to on output when you call the CLP executable. Note that you can turn the display output of the CLP to on by passing the `+o` or the `-o-` option to the SQLDB2.

Because the only REXX variable that is set after you call SQLDB2 is the SQLCA, you only use this routine to call DB2 APIs that do not return any data other than the SQLCA and that are not currently implemented through the SQLDBS interface. Thus, only the following DB2 APIs are supported by SQLDB2:

- Activate Database
- Add Node
- Bind for DB2 Version 1(1) (2)
- Bind for DB2 Version 2 or 5(1)
- Create Database at Node
- Drop Database at Node
- Drop Node Verify
- Deactivate Database
- Deregister
- Load(3)
- Load Query
- Precompile Program(1)
- Rebind Package(1)
- Redistribute Database Partition Group
- Register
- Start Database Manager
- Stop Database Manager

Notes on DB2 APIs Supported by SQLDB2:

1. These commands require a CONNECT statement through the SQLDB2 interface. Connections using the SQLDB2 interface are not accessible to the SQLEXEC interface and connections using the SQLEXEC interface are not accessible to the SQLDB2 interface.
2. Is supported on Windows®-based platforms through the SQLDB2 interface.
3. The optional output parameter, `pLoadInfoOut` for the Load API is not returned to the application in REXX.

Note: Although the SQLDB2 routine is intended to be used only for the DB2 APIs listed above, it can also be used for other DB2 APIs that are not supported through the SQLDBS routine. Alternatively, the DB2 APIs can be accessed through the CLP from within the REXX application.

Calling Stored Procedures from REXX

The sections that follow describe how to call stored procedures from REXX applications.

Stored Procedures in REXX

REXX SQL applications can call stored procedures at the database server by using the SQL CALL statement. The stored procedure can be written in any language supported on that server, except for REXX on AIX® systems. (Client applications may be written in REXX on AIX systems, but, as with other languages, they cannot call a stored procedure written in REXX on AIX.)

Related concepts:

• "Stored Procedure Calls in REXX" on page 507

Stored Procedure Calls in REXX

The CALL statement allows a client application to pass data to, and receive data from, a server stored procedure. The interface for both input and output data is a list of host variables. Because REXX generally determines the type and size of host variables based on their content, any output-only variables passed to CALL should be initialized with *dummy* data similar in type and size to the expected output.

Data can also be passed to stored procedures through SQLDA REXX variables, using the USING DESCRIPTOR syntax of the CALL statement. The following table shows how the SQLDA is set up. In the table, ':value' is the stem of a REXX host variable that contains the values needed for the application. For the DESCRIPTOR, 'n' is a numeric value indicating a specific *sqlvar* element of the SQLDA. The numbers on the right refer to the notes following the table.

Table 81. Client-side REXX SQLDA for Stored Procedures using the CALL Statement

USING DESCRIPTOR	:value.SQLD	1	
	:value.n.SQLTYPE	1	
	:value.n.SQLLEN	1	
	:value.n.SQLDATA	1	2
	:value.n.SQLDIND	1	2

Notes:

1. Before invoking the stored procedure, the client application must initialize the REXX variable with appropriate data.

   When the SQL CALL statement is executed, the database manager allocates storage and retrieves the value of the REXX variable from the REXX variable pool. For an SQLDA used in a CALL statement, the database manager allocates storage for the SQLDATA and SQLIND fields based on the SQLTYPE and SQLLEN values.

   In the case of a REXX stored procedure (that is, the procedure being called is itself written in Windows®-based REXX), the data passed by the client from

either type of CALL statement or the DARI API is placed in the REXX variable pool at the database server using the following predefined names:

SQLRIDA
Predefined name for the REXX input SQLDA variable

SQLRODA
Predefined name for the REXX output SQLDA variable

2. When the stored procedure terminates, the database manager also retrieves the value of the variables from the stored procedure. The values are returned to the client application and placed in the client's REXX variable pool.

Related concepts:

- "Client Considerations for Calling Stored Procedures in REXX" on page 508
- "Server Considerations for Calling Stored Procedures in REXX" on page 508
- "Retrieval of Precision and SCALE Values from SQLDA Decimal Fields" on page 508

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*

Client Considerations for Calling Stored Procedures in REXX

When using host variables in the CALL statement, initialize each host variable to a value that is type compatible with any data that is returned to the host variable from the server procedure. You should perform this initialization even if the corresponding indicator is negative.

When using descriptors, SQLDATA must be initialized and contain data that is type compatible with any data that is returned from the server procedure. You should perform this initialization even if the SQLIND field contains a negative value.

Related reference:

- "Supported SQL Data Types in REXX" on page 502

Server Considerations for Calling Stored Procedures in REXX

Ensure that all the SQLDATA fields and SQLIND (if it is a nullable type) of the predefined output sqlda SQLRODA are initialized. For example, if SQLRODA.SQLD is 2, the following fields must contain some data (even if the corresponding indicators are negative and the data is not passed back to the client):

- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

Retrieval of Precision and SCALE Values from SQLDA Decimal Fields

To retrieve the precision and scale values for decimal fields from the SQLDA structure returned by the database manager, use the `sqllen.scale` and `sqllen.precision` values when you initialize the SQLDA output in your REXX program. For example:

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485           /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale  = 2       /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision  = 7   /* WIDTH OF DECIMAL  */
io_sqlda.1.sqldata = 00000.00      /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1             /* NO INPUT DATA */
.
.
.
```

Chapter 24. Writing Applications Using DB2 WebSphere MQ Functions

WebSphere MQ Functional Overview

WebSphere® MQ provides the ability for message operations and database operations to be combined, and in some cases in a single unit of work as an atomic transaction. This feature is supported by the WebSphere MQ functions on UNIX® and Windows® with the non-transactional and transactional MQ user-defined functions, using schemas DB2MQ and DB2MQ1C.

A set of WebSphere MQ functions are provided with DB2® to allow SQL statements to include messaging operations. This means that this support is available to applications written in any supported language, for example, C, Java™, SQL using any of the database interfaces. All examples shown below are in SQL. This SQL can be used from other programming languages in all of the standard ways. All of the WebSphere MQ messaging styles described above are supported.

In a basic configuration, a WebSphere MQ server is located on the database server machine along with DB2. The WebSphere MQ functions are installed into DB2 and provide access to the WebSphere MQ server. DB2 clients can be located on any machine accessible to the DB2 server. Multiple clients can concurrently access the WebSphere MQ functions through the database. Through the provided functions, DB2 clients can perform messaging operations within SQL statements. These messaging operations allow DB2 applications to communicate among themselves or with other WebSphere MQ applications.

The **enable_MQFunctions** command is used to enable a DB2 database for the WebSphere MQ functions. It will automatically establish a simple default configuration that client applications may utilize with no further administrative action. For a description, see the topics ″enable_MQFunctions″ and ″disable_MQFunctions″. The default configuration allows application programmers a quick way to get started and a simpler interface for development. Additional functionality can be configured incrementally as needed.

To send a simple message using the default configuration, use the following SQL statement:

```
Example 1:
VALUES DB2MQ.MQSEND('simple message')
```

This sends the message “simple message” to the WebSphere MQ queue manager and queue specified by the default configuration.

The Application Messaging Interface (AMI) of WebSphere MQ provides a clean separation between messaging actions and the definitions that dictate how those actions should be carried out. These definitions are kept in an external repository file and managed using the AMI Administration tool. This makes AMI applications

simple to develop and maintain. The WebSphere MQ functions provided with DB2 Universal Database™ are based on the AMI WebSphere MQ interface. AMI supports the use of an external configuration file, called the AMI Repository, to store configuration information. The default configuration includes a WebSphere MQ AMI Repository configured for use with DB2 Universal Database.

Two key concepts in WebSphere MQ AMI, service points and policies, are carried forward into the DB2 WebSphere MQ functions. A service point is a logical end-point from which a message can be sent or received. In the AMI repository, each service point is defined with a WebSphere MQ queue name and queue manager. Policies define the quality of service options that should be used for a given messaging operation. Key qualities of service include message priority and persistence. Default service points and policy definitions are provided and can be used by developers to further simplify their applications. The example in "Example 1" on page 511 can be re-written as follows to explicitly specify the default service point and policy name:

Example 2:

```
VALUES DB2MQ.MQSEND('DB2.DEFAULT.SERVICE',
                    'DB2.DEFAULT.POLICY',
                    'simple message')
```

Queues can be serviced by one or more applications at the server upon which the queues and applications reside. In many configurations multiple queues are defined to support different applications and purposes. For this reason, it is often important to define different service points when making WebSphere MQ requests. This is demonstrated in the following example:

Example 3:

```
VALUES DB2MQ.MQSEND('ODS_Input', 'simple message')
```

In the above example, the policy is not specified and so the default policy is used.

Limitations

When using the sending or receiving functions, the maximum length of a message of type VARCHAR is 4000 characters for schema DB2MQ, and 32,000 characters for schema DB2MQ1C. The maximum length when sending or receiving a message of type CLOB is 1 MB for schema DB2MQ. These are also the maximum message sizes for publishing a message using MQPublish.

Different functions are sometimes required when working with CLOB messages and VARCHAR messages. Generally, the CLOB version of an MQ function uses the identical syntax as its counterpart. The only difference is that its name has the characters CLOB at the end. For example, the CLOB equivalent of MQREAD is MQREADCLOB.

Error codes

The WebSphere MQ function return codes can be found in Appendix B of the MQSeries® Application Messaging Interface Manual. Other messages, using component identifier AMI, are listed in the DB2 Universal Database Messages and Codes.

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Messaging" on page 513
- "Sending Messages with WebSphere MQ Functions" on page 515

- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

WebSphere MQ Messaging

The DB2® WebSphere® MQ functions support three messaging models: datagrams, publish/subscribe (p/s), and request/reply (r/r). Messages that are sent as datagrams are sent to a single destination with no reply expected. In the p/s model, one or more publishers send a message to a publication service which distributes the message to one or more subscribers. Request/reply is similar to datagram, but the sender expects to receive a response.

WebSphere MQ does not, itself, mandate or support any particular structuring of the messages it transports. Other products, such as MQSeries® Integrator (MQSI) do offer support for messages formed as C or Cobol or as XML strings. Structured messages in MQSI are defined by a message repository. XML messages typically have a self-describing message structure and can also be managed through the repository. Messages can also be unstructured, requiring user code to parse or construct the message content. Such messages are often semi-structured, that is, they use either byte positions or fixed delimiters to separate the fields within a message. Support for such semi-structured messages is provided by the WebSphere MQ Assist Wizard. Support for XML messages is provided through features of the DB2 XML Extender.

The most basic form of messaging with the WebSphere MQ DB2 functions occurs when all database applications connect to the same DB2 server. Clients can be local to the database server or distributed in a network environment.

In a simple scenario, Client A invokes the MQSEND function to send a user-defined string to the default service location. The WebSphere MQ functions are then executed within DB2 on the database server. At some later time, Client B invokes the MQRECEIVE function to remove the message at the head of the queue defined by the default service and return it to the client. Again, the WebSphere MQ functions used to perform this work are executed by DB2.

Database clients can use simple messaging in a number of ways. Some common uses for messaging are:

Data collection
Information is received in the form of messages from one or more possibly diverse sources of information. Information sources can be commercial applications such as SAP or applications developed in-house. Such data can be received from queues and stored in database tables for further processing or analysis.

Workload distribution
Work requests are posted to a queue shared by multiple instances of the same application. When an instance is ready to perform some work, it receives a message from the top of the queue containing a work request to perform. Using this technique, multiple instances can share the workload represented by a single queue of pooled requests.

Application signaling
In a situation where several processes collaborate, messages are often used to coordinate their efforts. These messages can contain commands or requests for work to be performed. Typically, this kind of signaling is one-way; that is, the party that initiates the message does not expect a reply. See the topic ″Request/Reply Communications with WebSphere MQ Functions″ for more information.

Application notification
Notification is similar to signaling in that data is sent from an initiator with no expectation of a response. Typically, however, notification contains data about business events that have taken place. Publish/Subscribe is a more advanced form of notification. For more information, see the topic ″Publish/Subscribe with WebSphere MQ Functions″.

The following scenario extends the simple scenario described above to incorporate remote messaging. That is, a message is sent between Machine A and Machine B. The sequence of steps is as follows:

1. The DB2 Client executes an MQSEND call, specifying a target service that has been defined to represent a remote queue on Machine B.
2. The WebSphere MQ DB2 functions perform the actual WebSphere MQ work to send the message. The WebSphere MQ server on Machine A accepts the message and guarantees that it will deliver it to the destination defined by the service point definition and current WebSphere MQ configuration of Machine A. The server determines that this is a queue on Machine B. It then attempts to deliver the message to the WebSphere MQ server on Machine B, transparently retrying as needed.
3. The WebSphere MQ server on Machine B accepts the message from the server on Machine A and places it in the destination queue on Machine B.

4. A WebSphere MQ client on Machine B requests the message at the head of the queue.

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "Sending Messages with WebSphere MQ Functions" on page 515
- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

Sending Messages with WebSphere MQ Functions

By using MQSEND, a DB2® user chooses what data to send, where to send it, and when it will be sent. In the industry this is commonly called "Send and Forget"" meaning that the sender just sends a message, relying on the guaranteed delivery protocols of WebSphere® MQ to ensure that the message reaches its destination. The following examples illustrate this.

"Example 4" sends a user-defined string to the service point myPlace with the policy highPriority:

```
Example 4:
VALUES DB2MQ.MQSEND('myplace','highPriority','test')
```

In "Example 4" on page 515, the policy highPriority refers to a policy defined in the AMI Repository that sets the WebSphere MQ priority to the highest level and perhaps adjusts other qualities of service, such as persistence, as well.

The message content can be composed of any legal combination of SQL and user-specified data. This includes nested functions, operators, and casts. For instance, given a table EMPLOYEE, with VARCHAR columns LASTNAME, FIRSTNAME, and DEPARTMENT, if you want to send a message containing this information for each employee in DEPARTMENT 5LGA you would do the following:

```
Example 5:
SELECT DB2MQ.MQSEND(LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT)
  FROM EMPLOYEE WHERE DEPARTMENT = '5LGA'
```

If this table also had a column AGE defined as an integer, you could include it with the following addition to the statement:

```
Example 6:
SELECT DB2MQ.MQSEND (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT ||
  ' ' || char(AGE)) FROM EMPLOYEE WHERE DEPARTMENT = '5LGA'
```

If the table EMPLOYEE has a column RESUME of type CLOB instead of an AGE column, then a message containing the information for each employee in DEPARTMENT 5LGA can be issued with the following statement:

```
Example 7:
SELECT DB2MQ.MQSEND(clob(LASTNAME) || ' ' || clob(FIRSTNAME) ||
  ' ' || clob(DEPARTMENT) || ' ' || RESUME))
  FROM EMPLOYEE WHERE DEPARTMENT = '5LGA'
```

The following example shows how message content can be derived using any valid SQL expression. Given a second table DEPT containing VARCHAR columns DEPT_NO and DEPT_NAME, messages can be sent that contain employee LASTNAME and DEPT_NAME:

```
Example 8:
SELECT DB2MQ.MQSEND(e.LASTNAME || ' ' || d.DEPTNAME)
  FROM EMPLOYEE e, DEPT d
  WHERE e.DEPARTMENT = d.DEPTNAME
```

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "WebSphere MQ Messaging" on page 513
- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*

- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

Retrieving Messages with WebSphere MQ Functions

By using the WebSphere® MQ DB2® functions, messages can be either received or read. The difference between reading and receiving is that reading returns the message at the head of a queue without removing it from the queue, while receiving operations cause the message to be removed from the queue. When you use a receive operation to retrieve a message, the same message can only be retrieved once. When you use a read operation to retrieve a message, the same message can be retrieved many times. The following examples demonstrate the retrieve operations.

Example 9:
```
VALUES DB2MQ.MQREAD()
```

"Example 9" on page 517 returns a VARCHAR string containing the message at the head of queue that is defined by the default service using the default quality of service policy. If no messages are available to be read, a null value is returned. This operation does not change the queue.

Example 10:
```
VALUES DB2MQ.MQRECEIVE('Employee_Changes')
```

"Example 10" on page 517 shows how a message can be removed from the head of the queue defined by the Employee_Changes service using the default policy.

One feature of DB2 is the ability to generate a table from a user-defined (or DB2–provided) function. You can exploit this table-function feature so that the contents of a queue are materialized as a DB2 Universal Database™ table. The following example demonstrates the simplest form of this feature:

Example 11:
```
SELECT t.* FROM table ( DB2MQ.MQREADALL()) t
```

The query in "Example 11" returns a table consisting of all of the messages in the queue that are defined by the default service and the metadata about these messages.

To return just the messages, the example could be rewritten:

Example 12:
```
SELECT t.MSG FROM table (DB2MQ.MQREADALL()) t
```

The table returned by a table function is no different from a table retrieved from the database directly. This means that you can use this table in a wide variety of ways. For instance, you can join the contents of the table with another table or count the number of messages in a queue:

Example 13:

```
SELECT t.MSG, e.LASTNAME FROM table (DB2MQ.MQREADALL() ) t, EMPLOYEE e
  WHERE t.MSG = e.LASTNAME
```

Example 14:

```
SELECT COUNT(*) FROM table (DB2MQ.MQREADALL()) t
```

You can also hide the fact that the source of the table is a queue by creating a view over a table function. For instance, the following example creates a view called NEW_EMP over the queue referred to by the service named NEW_EMPLOYEES:

Example 15:

```
CREATE VIEW NEW_EMP (msg) AS SELECT t.msg
   FROM table(DB2MQ.MQREADALL(NEW_EMPLOYEES)) t
```

In this case ("Example 15"), the view is defined with only a single column containing an entire message. If messages are simply structured, for instance containing two fields of fixed length, it is straightforward to use the DB2 built-in functions to parse the message into the two columns. For example, if you know that messages sent to a particular queue always contain an 18-character last name followed by an 18-character first name, then you can define a view containing each field as a separate column as follows:

Example 16:

```
CREATE VIEW NEW_EMP2 AS SELECT left(t.msg,18) AS LNAME, right(t.msg,18) AS FNAME
   FROM table(DB2MQ.MQREADALL()) t
```

A feature of the DB2 Development Center, the WebSphere MQ Assist Wizard, can be used to create new DB2 table functions and views that map delimited message structures to columns.

It is often desirable to store the contents of one or more messages into the database. This can be done by using the full power of SQL to manipulate and store message content. A simple example of this is shown below:

Example 17:

```
INSERT INTO MESSAGES SELECT t.msg FROM table (DB2MQ.MQRECEIVEALL()) t
```

Given a table MESSAGES, with a single column of VARCHAR(2000), the above statement inserts the messages from the default service queue into the table. This technique can be used to cover a very wide variety of circumstances.

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "WebSphere MQ Messaging" on page 513
- "Sending Messages with WebSphere MQ Functions" on page 515
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

WebSphere MQ Application-to-application Connectivity

Application integration is a common element in many solutions. Whether integrating a purchased application into an existing infrastructure or just integrating a newly developed application into an existing environment, we are often faced with the task of gluing a heterogeneous collection of subsystems together to form a working whole.

WebSphere® MQ is commonly viewed as an essential tool for integrating applications. Accessible in most hardware, software, and language environments, WebSphere MQ provides the means to interconnect a very heterogeneous collection of applications.

Please see the topics ″Request/Reply Communications with WebSphere MQ Functions″ and ″Publish/Subscribe with WebSphere MQ Functions″ for application integration scenarios and how they can be used with DB2®.

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "WebSphere MQ Messaging" on page 513
- "Sending Messages with WebSphere MQ Functions" on page 515
- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

Request/Reply Communications with WebSphere MQ Functions

The Request/Reply (R/R) communications method is a very common technique for one application to request the services of another. One way to do this is for the requester to send a message to the service provider requesting some work to be performed. Once the work has been completed, the provider may decide to send results (or just a confirmation of completion) back to the requestor. However, using the basic messaging techniques described above, there is nothing that connects the sender's request with the service provider's response. Unless the requester waits for a reply before continuing, some mechanism must be used to associate each reply with its request. Rather than force the developer to create such a mechanism, WebSphere® MQ provides a correlation identifier that allows the correlation of messages in an exchange.

While there are a number of ways in which this mechanism could be used, the simplest is for the requestor to mark a message with a known correlation identifier using, for instance, the following:

Example 18:

```
VALUES DB2MQ.MQSEND ('myRequester','myPolicy','SendStatus:cust1','Req1')
```

This statement adds a final parameter Req1 to the MQSEND statement from above to indicate the correlation identifier for the request. To receive a reply to this specific request, use the corresponding MQRECEIVE statement to selectively retrieve the first message defined by the indicated service that matches this correlation identifier as follows:

Example 19:

```
VALUES DB2MQ.MQRECEIVE('myReceiver','myPolicy','Req1')
```

If the application servicing the request is busy and the requestor issues the above MQRECEIVE before the reply is sent, then no messages matching this correlation identifier are found.

To receive both the service request and the correlation identifier, use a statement like the following:

```
Example 20:
SELECT msg, correlid
  FROM table (VALUES DB2MQ.MQRECEIVEALL('aServiceProvider','myPolicy',1)) t
```

This returns the message and correlation identifier of the first request from the service aServiceProvider.

Once the service has been performed, it sends the reply message to the queue described by aRequester. Meanwhile, the service requester could have been doing other work. In fact, there is no guarantee that the initial service request will receive a response within a set time. Application level timeouts such as this must be managed by the developer; the requester must poll to detect the presence of the reply. The advantage of such time-independent asynchronous processing is that the requester and service provider execute completely independently of one another. This can be used both to accommodate environments in which applications are only intermittently connected, and more batch-oriented environments in which multiple requests or replies are aggregated before processing. This kind of aggregation is often used in data warehouse environments to periodically update a data warehouse or operational data store.

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "WebSphere MQ Messaging" on page 513
- "Sending Messages with WebSphere MQ Functions" on page 515
- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Publish/Subscribe with WebSphere MQ Functions" on page 522
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*
- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*

- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

Publish/Subscribe with WebSphere MQ Functions

Two techniques in database messaging are simple data publication and automated publication.

Simple Data Publication

A common scenario in application integration is for one application to notify other applications about events of interest. This is easily done by sending a message to a queue monitored by another application. The contents of the message can be a user-defined string or can be composed from database columns. Often a simple message is all that needs to be sent using the MQSEND function. When such messages need to be sent concurrently to multiple recipients, the Distribution List facility of the WebSphere® MQ AMI can be used.

A distribution list is defined using the AMI Administration tool. A distribution list comprises a list of individual services. A message sent to a distribution list is forwarded to every service defined within the list. This is especially useful when it is known that a few services will always be interested in every message. The following example shows sending of a message to the distribution list interestedParties:

```
Example 21:
VALUES DB2MQ.MQSEND('interestedParties', 'information of general interest');
```

When more control over the messages that particular services should receive is required, a Publish/Subscribe capability is needed. Publish/Subscribe systems typically provide a scalable, secure environment in which many subscribers can register to receive messages from multiple publishers. To support this capability, the MQPublish interface can be used, in conjunction with MQSeries® Integrator or the WebSphere MQ Publish/Subscribe facility.

MQPublish allows users to optionally specify a topic to be associated with a message. Topics allow a subscriber to more clearly specify the messages to be accepted. The sequence of steps is as follows:

1. A WebSphere MQ administrator configures MQSeries Integrator publish/subscribe capabilities.
2. Interested applications subscribe to subscription points defined by the MQSI configuration, optionally specifying topics of interest to them. Each subscriber selects relevant topics, and can also utilize the content-based subscription techniques of MQSeries Integrator Version 2. It is important to note that queues, as represented by service names, define the subscriber.
3. A DB2® application publishes a message to the service point Weather. The messages indicate that the weather is Sleet with a topic of Austin, thus notifying interested subscribers that the weather in Austin is Sleet.
4. The mechanics of actually publishing the message are handled by the WebSphere MQ functions provided by DB2. The message is sent to MQSeries Integrator using the service named Weather.
5. MQSI accepts the message from the Weather service, performs any processing defined by the MQSI configuration, and determines which subscriptions satisfy. MQSI then forwards the message to the subscriber queues whose criteria it meets.

6. Applications that have subscribed to the Weather service, and registered an interest in Austin, will receive the message Sleet in their receiving service.

To publish this data using all the defaults and a null topic, use the following statement:

Example 22:
```
SELECT DB2MQ.MQPUBLISH (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT
  ||' ' || char(AGE)) FROM EMPLOYEE WHERE DEPARTMENT = '5LGA'
```

Fully specifying all the parameters and simplifying the message to contain only the LASTNAME, the statement now looks like:

Example 23:
```
SELECT DB2MQ.MQPUBLISH('HR_INFO_PUB', 'SPECIAL_POLICY', LASTNAME,
  'ALL_EMP:5LGA', 'MANAGER') FROM EMPLOYEE WHERE DEPARTMENT = '5LGA'
```

This statement publishes messages to the HR_INFO_PUB publication service using the SPECIAL_POLICY service. The messages indicate that the sender is the MANAGER correlation identifier. The topic string ("ALL_EMP:5LGA") demonstrates that you can specify multiple topics, concatenating them using a colon (":"). In this example, the use of two topics allows subscribers to register for either ALL_EMP or just 5LGA to receive these messages. To receive published messages, you must first register your interest in messages containing a given topic and indicate the name of the subscriber service that messages should be sent to. It is important to note that an AMI subscriber service defines a broker service and a receiver service. The broker service is how the subscriber communicates with the publish/subscribe broker and the receiver service is where messages matching the subscription request will be sent. The following statement registers an interest in the topic ALL_EMP.

Example 24:
```
VALUES DB2MQ.MQSUBSCRIBE('aSubscriber', 'ALL_EMP')
```

Once an application has subscribed, messages published with the topic ALL_EMP are forwarded to the receiver service that is defined by the subscriber service. An application can have multiple concurrent subscriptions. To obtain the messages that meet your subscription, you can use any of the standard message retrieval functions. For instance, if the subscriber service aSubscriber defines the receiver service to be aSubscriberReceiver, then the following statement non-destructively reads the first message:

Example 25:
```
VALUES DB2MQ.MQREAD('aSubscriberReceiver')
```

To determine both the messages and the topics that they were published under, use one of the table functions. The following statement receives the first five messages from aSubscriberReceiver and displays both the message and the topic:

Example 26:
```
SELECT t.msg, t.topic FROM table (DB2MQ.MQRECEIVEALL('aSubscriberReceiver',5)) t
```

To read all of the messages with the topic ALL_EMP, you can leverage the power of SQL by issuing the following statement:

Example 27:
```
SELECT t.msg FROM table (DB2MQ.MQREADALL('aSubscriberReceiver')) t
  WHERE t.topic = 'ALL_EMP'
```

Note: It is important to realize that if MQRECEIVEALL is used with a constraint then the entire queue will be consumed, not just those messages published with topic ALL_EMP. This is because the table function is performed before the constraint is applied.

When you are no longer interested in subscribing to a particular topic you must explicitly unsubscribe using a statement such as:

Example 28:

```
VALUES DB2MQ.MQUNSUBSCRIBE('aSubscriber', 'ALL_EMP')
```

After you issue the above statement, the publish/subscribe broker no longer delivers messages matching this subscription.

Automated Publication

Another important technique in database messaging is automated publication. Using the trigger facility within DB2, you can automatically publish messages as part of a trigger invocation. While other techniques exist for automated data publication, the trigger-based approach allows administrators or developers great freedom in constructing the message content and flexibility in defining the trigger actions. As with any use of triggers, you must be aware of the frequency and cost of execution. The following examples demonstrate how triggers can be used with the WebSphere MQ DB2 functions.

The example below shows how easy it is to publish a message each time a new employee is hired. Any users or applications subscribing to the HR_INFO_PUB service with a registered interest in NEW_EMP will receive a message containing the date, name and department of each new employee.

Example 29:

```
CREATE TRIGGER new_employee AFTER INSERT ON employee
  REFERENCING NEW AS n FOR EACH ROW MODE DB2SQL
  VALUES DB2MQ.MQPUBLISH('HR_INFO_PUB',current date||''||
    'LASTNAME'||''DEPARTMENT,'NEW_EMP')
```

Related concepts:

- "MQSeries Enablement" on page 16
- "WebSphere MQ Functional Overview" on page 511
- "WebSphere MQ Messaging" on page 513
- "Sending Messages with WebSphere MQ Functions" on page 515
- "Retrieving Messages with WebSphere MQ Functions" on page 517
- "WebSphere MQ Application-to-application Connectivity" on page 519
- "Request/Reply Communications with WebSphere MQ Functions" on page 520
- "How to use WebSphere MQ functions within DB2" in the *IBM DB2 Information Integrator Application Developer's Guide*

Related tasks:

- "Setting up DB2 WebSphere MQ functions" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "MQSEND scalar function" in the *SQL Administrative Routines*
- "MQRECEIVE scalar function" in the *SQL Administrative Routines*
- "MQREAD scalar function" in the *SQL Administrative Routines*
- "MQPUBLISH scalar function" in the *SQL Administrative Routines*
- "MQSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQUNSUBSCRIBE scalar function" in the *SQL Administrative Routines*
- "MQREADALL table function" in the *SQL Administrative Routines*

- "MQRECEIVEALL table function" in the *SQL Administrative Routines*
- "MQREADCLOB scalar function" in the *SQL Administrative Routines*
- "MQRECEIVECLOB scalar function" in the *SQL Administrative Routines*
- "MQREADALLCLOB table function" in the *SQL Administrative Routines*
- "MQRECEIVEALLCLOB table function" in the *SQL Administrative Routines*
- "db2mqlsn - MQ Listener Command" in the *Command Reference*
- "enable_MQFunctions" in the *Command Reference*
- "disable_MQFunctions" in the *Command Reference*

Chapter 25. WebSphere

The sections that follow describe WebSphere connection pooling and statement caching.

Connections to Enterprise Data

Many companies store their data on large systems such as the zSeries® servers. Web applications need ways to get to that data.

DB2® Connect gives a Windows®-based application or UNIX®-application the ability to connect to and use the data stored on these large systems. DB2 also provides its own set of features such as connection pooling, and the connection concentrator.

WebSphere Connection Pooling and Data Sources

Each time a resource attempts to access a database, it must connect to that database. A database connection incurs overhead; it requires resources to create the connection, maintain it, and then release it when it is no longer required.

Note: The information provided here refers to Version 4 of the WebSphere® Application Server for UNIX®, LINUX, and Windows®.

The total database overhead for an application is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter, because of the nature of the Internet. Often, more effort is spent connecting and disconnecting than is spent during the interactions themselves. Further, because Internet requests can arrive from virtually anywhere, usage volumes can be large and difficult to predict.

IBM® WebSphere Application Server enables administrators to establish a pool of database connections that can be shared by applications on an application server to address these overhead problems.

Connection pooling spreads the connection overhead across several user requests, thereby conserving resources for future requests.

You can either use WebSphere connection pooling, or you can use the DB2® connection pooling support that is provided by the JDBC 2.1 Optional Package API to establish the connection pool.

WebSphere connection pooling is the implementation of the JDBC 2.1 Optional Package API specification. Therefore, the connection pooling programming model is as specified in the JDBC 2.1 Core and JDBC 2.1 Optional Package API specifications. This means that applications obtaining their connections through a datasource created in WebSphere Application Server can benefit from JDBC 2.1 features such as pooling of connections and JTA-enabled connections.

In addition, WebSphere connection pooling provides additional features that enable administrators to tune the pool for best performance and provide applications with WebSphere exceptions that enable programmers to write applications without

knowledge of common vendor-specific SQLExceptions. Not all vendor-specific SQLExceptions are mapped to WebSphere exceptions; applications must still be coded to deal with vendor-specific SQLExceptions. However, the most common, recoverable exceptions are mapped to WebSphere exceptions.

The datasource obtained through WebSphere is a datasource that implements the JDBC 2.1 Optional Package API. It provides pooling of connections and, depending on the vendor-specific datasource selected, may provide connections capable of participating in two-phase commit protocol transactions (JTA-enabled).

The `AccessEmployee` program in the `AccessEmployee.ear` file uses the WebSphere `DataSource` to access a DB2 database.

Related reference:

- "Java WebSphere samples" in the *Application Development Guide: Building and Running Applications*

Benefits of WebSphere Connection Pooling

Connection pooling can improve the response time of any application that requires connections, especially Web-based applications.

When a user makes a request over the Web to a resource, the resource accesses a datasource. Most user requests do not incur the overhead of creating a new connection, because the datasource might locate and use an existing connection from the pool of connections. When the request is satisfied and the response is returned to the user, the resource returns the connection to the connection pool for reuse. Again, the overhead of a disconnect is avoided.

Each user request incurs a fraction of the cost of connection or disconnecting. After the initial resources are used to produce the connections in the pool, additional overhead is insignificant because the existing connections are reused.

Caching of prepared statements is another mechanism by which WebSphere® connection pooling improves Web-based application response times.

A cache of previously prepared statements is available on a connection. When a new prepared statement is requested on a connection, the cached prepared statement is returned if available. This caching reduces the number of costly prepared statements created, which improves response times. The cache is useful for applications that tend to prepare the same statement time and again.

In addition to improved response times, WebSphere connection pooling provides a layer of abstraction from the database which can buffer the client application and make different databases appear to work in the same manner to an application

This buffering makes it easier to switch application databases, because the application code does not have to deal with common vendor-specific SQLExceptions but, rather, with a WebSphere connection pooling exception.

Statement Caching in WebSphere

WebSphere® provides a mechanism for caching previously prepared statements. Caching prepared statements improves response times, because an application can reuse a `PreparedStatement` on a connection if it exists in that connection's cache, bypassing the need to create a new `PreparedStatement`.

When an application creates a `PreparedStatement` on a connection, the connection's cache is first searched to determine if a `PreparedStatement` with the same SQL string already exists. This search is done by using the entire string of SQL statements in the `prepareStatement()` method. If a match is found, the cached PreparedStatement is returned for use. If it is not, a new `PreparedStatement` is created and returned to the application.

As the prepared statements are closed by the application, they are returned to the connection's cache of statements. By default, only 100 prepared statements can be kept in cache for the entire pool of connections. For example, if there are ten connections in the pool, the number of cached prepared statements for those ten connections cannot exceed 100. This ensures that a limited number of prepared statements are concurrently open to the database, which helps to avoid resource problems with a database.

Elements are removed from the connection's cache of prepared statements only when the number of currently cached prepared statements exceeds the `statementCacheSize` (by default 100). If a prepared statement needs to be removed from the cache, it is removed and added to a vector of discarded statements. As soon as the method in which the prepared statement was removed has ended, the prepared statements on the discarded statements vector are closed to the database. Therefore, at any given time, there might be 100 plus the number of recently discarded statements open to the database. The extra prepared statements are closed after the method ends.

The number of prepared statements to be cached is configurable at the data source. Each cache should be tuned according to the application's requirements for prepared statements.

Part 6. Security Plug-ins

Chapter 26. Security plug-ins

Security plug-ins

Authentication in DB2® is done using *security plug-ins*. A security plug-in is a dynamically loadable library that DB2 loads to provide the following functionality:

- group retrieval plug-in: retrieves group membership information for a given user
- client authentication plug-in: manages authentication on a DB2 client.
- server authentication plug-in: manages authentication on a DB2 server.

DB2 supports two mechanisms for plug-in authentication:

- Authentication using a user ID and password which is known as user ID/password authentication. The authentication types CLIENT, SERVER, SERVER_ENCRYPT, DATA_ENCRYPT, and DATA_ENCRYPT_CMP determine how and where authentication of a user takes place. The authentication type used depends on the authentication type specified using the *authentication* database configuration parameter. These authentication types are all implemented using user ID/password authentication plug-ins.
- Authentication using GSS-API, formally known as *Generic Security Service Application Program Interface, Version 2* (IETF RFC2743) and *Generic Security Service API Version 2: C-Bindings* (IETF RFC2744). Kerberos authentication is also implemented using GSS-API. The authentication types KERBEROS, GSSPLUGIN, KRB_SERVER_ENCRYPT, and GSS_SERVER_ENCRYPT use GSS-API authentication plug-ins. KRB_SERVER_ENCRYPT and GSS_SERVER_ENCRYPT support both GSS-API authentication and user ID/password authentication however GSS-API authentication is the preferred authentication type.

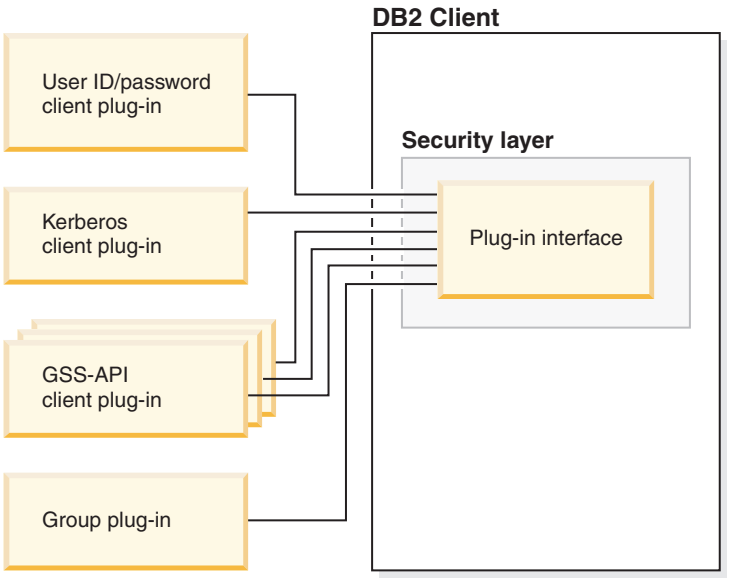
Each of the plug-ins can be used independently or in conjunction with one or more of the other plug-ins. For example you might only use a server authentication plug-in and assume the DB2 defaults for client and group authentication. Alternatively you might have only a group or client authentication plug-in. The only case however where both a client and server plug-in are required is for GSS-API authentication plug-ins.

In DB2 Version 8.2, the default behavior is to use a user ID/password plug-in that implements an operating system level mechanism for authentication. In all previous releases of DB2 the default behavior is to directly use operating system level authentication without a plug-in implementation. In DB2 Version 8.2 client-side Kerberos support is available on Solaris, AIX®, Windows®, and IA32 Linux operating systems however it is only enabled by default on Windows.

DB2 includes sets of plug-ins for group retrieval, user ID/password authentication, and for Kerberos authentication. With the security plug-in architecture you can customize DB2's authentication behavior by either developing your own plug-ins or buying plug-ins from a third party.

Deployment of security plug-ins on DB2 clients:

DB2 clients can support one group plug-in, one user ID/password authentication plug-in, and will negotiate with the DB2 server for a particular GSS-API plug-in. This negotiation consists of a scan by the client of the DB2 server's list of implemented GSS-API plug-ins for the first authentication plug-in name that matches an authentication plug-in implemented on the client. The server's list of plug-ins is a user-specified database manager configuration parameter value that contains the names of all of the plug-ins that are implemented on the server. The following figure portrays the security plug-in infrastructure on a DB2 client.

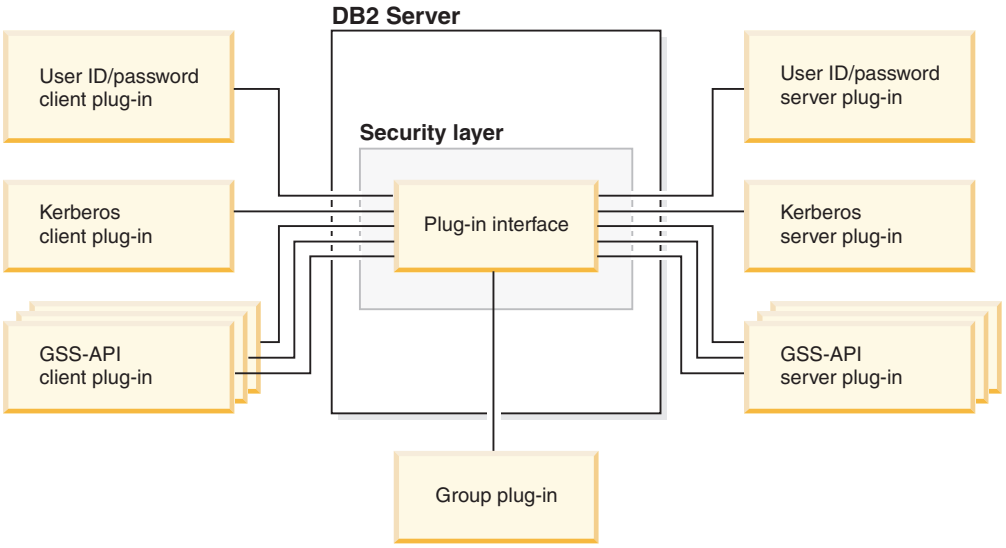


Deployment of security plug-ins on DB2 servers:

DB2 servers can support one group plug-in, one user ID/password authentication plug-in, and multiple GSS-API plug-ins. The multiple GSS-API plug-ins are named in a database manager configuration parameter value as a list. Only one GSS-API plug-in in this list can be a Kerberos plug-in.

In addition to deploying server-side security plug-ins, you might also need to deploy client authorization plug-ins on your database server. When you run instance-level operations like db2start and db2trc, DB2 performs authorization checking for the operation using client authentication plug-ins. Therefore, you should install a client version of the authentication plug-in of the type specified in the database manager configuration parameter *authentication*. If you do not deploy client authentication plug-ins on the database server, instance level operations such as db2start will fail. For example, if the authentication type is SERVER and no user-supplied client plug-in is used, then DB2 will fall back to using the IBM shipped default client operating system plug-in. The following figure portrays the

security plug-in infrastructure on a DB2 server.



Enabling security plug-ins:

The system administrator can specify the names of the plug-ins to use for each authentication mechanism by updating certain plug-in-related database manager configuration parameters. If these parameters are null, they will default to the DB2-supplied plug-ins for group retrieval, user ID/password management, or Kerberos (if authentication is set to Kerberos -- server-side only). However, DB2 does not provide a default GSS-API plug-in. Therefore, if the system administrator specifies an authentication type of GSSPLUGIN in authentication, she must also specify an authentication plug-in in *srvcon_gssplugin_list*.

How DB2 loads security plug-ins:

On the DB2 server, all of the supported plug-ins identified in the database manager configuration parameters will be loaded when the database manager starts up.

For database connections and instance attachments, the DB2 client will load an appropriate plug-in based on the security mechanism negotiated with the server during connection or attachment operations. It is possible that a client application can cause multiple security plug-ins to be concurrently loaded and in use. This could happen, for example, in a threaded program that has concurrent connections to different databases from different instances.

For actions other than database connections or instance attachments that require authorization (such as updating the database configuration, starting and stopping the database manager, turning DB2 trace on and off), the DB2 client program will load a plug-in specified in another database manager configuration parameter. If *authentication* is set to GSSPLUGIN, then DB2 will use the plug-in specified by *local_gssplugin*. If *authentication* is set to KERBEROS, then DB2 will use the plug-in specified by *clnt_krb_plugin*. Otherwise, DB2 will use the plug-in specified by *clnt_pw_plugin*.

Developing security plug-ins:

If you are developing a security plug-in, you will need to implement the standard authentication functions that DB2 will invoke. For the available types of plug-ins, the functionality you will need to implement is as follows:

Group retrieval
Get the list of groups to which a user belongs.

User ID/password authentication
Identify the default security context (client only), validate and optionally change a password, determine if a given string represents a valid user (server only), modify the user ID or password provided on the client before it is sent to the server (client only), return the DB2 authorization ID associated with a given user.

GSS-API authentication
Implement the required GSS-API functions, identify the default security context (client only), generate initial credentials based on a user ID and password and optionally change password (client only), create and accept security tickets, and return the DB2 authorization ID associated with a given GSS-API security context.

Related concepts:

- "Authentication methods for your server" in the *Administration Guide: Implementation*
- "Security plug-in library locations" on page 536
- "How DB2 loads security plug-ins" on page 549
- "Security plug-in APIs" on page 559

Related reference:

- "authentication - Authentication type configuration parameter" in the *Administration Guide: Performance*
- "srvcon_auth - Authentication type for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*
- "Security plug-in samples" in the *Application Development Guide: Building and Running Applications*

Security plug-in library locations

Once you acquire your security plug-ins (by developing them yourself, or purchasing them from a third party), you need to copy them to specific locations on your database server.

DB2® will look for client-side user authentication plug-ins in the following directory:

- UNIX® 32-bit: `$DB2PATH/security32/plugin/client`
- UNIX 64-bit: `$DB2PATH/security64/plugin/client`
- WINDOWS 32-bit and 64-bit: `$DB2PATH\security\plugin\<instance name>\client`

Note: For Windows®, the subdirectories <instance name> and client are not created automatically. The instance owner has to manually create them.

DB2 will look for server-side user authentication plug-ins in the following directory:

- UNIX 32-bit: `$DB2PATH/security32/plugin/server`
- UNIX 64-bit: `$DB2PATH/security64/plugin/server`
- WINDOWS 32-bit and 64-bit: `$DB2PATH\security\plugin\<instance name>\server`

Note: For Windows, the subdirectories <instance name> and server are not created automatically. The instance owner has to manually create them.

DB2 will look for group plug-ins in the following directory:

- UNIX 32-bit: `$DB2PATH/security32/plugin/group`
- UNIX 64-bit: `$DB2PATH/security64/plugin/group`
- WINDOWS 32-bit and 64-bit: `$DB2PATH\security\plugin\<instance name>\group`

Note: For Windows, the subdirectories <instance name> and group are not created automatically. The instance owner has to manually create them.

Related concepts:

- "Security plug-ins" on page 533
- "How DB2 loads security plug-ins" on page 549

Related tasks:

- "Deploying a group retrieval plug-in" on page 543
- "Deploying a user ID/password plug-in" on page 543
- "Deploying a GSS-API plug-in" on page 545
- "Deploying a Kerberos plug-in" on page 547

Related reference:

- "Restrictions on security plug-in libraries" on page 550

Security plug-in naming conventions

The security plug-in libraries must have the appropriate filename extension for each individual platform. By operating system these extensions are as follows:

- Windows®: .DLL
- AIX®: .a
- Linux, HP IPF and Solaris: .so
- HPUX on PA-RISC: .sl

For example, assume you have a security plug-in library called `MyPlugin`. For each supported operating system, the appropriate library file name follows:

- Windows 32-bit: MyPlugin.dll
- Windows 64-bit: MyPlugin64.dll
- AIX 32 or 64-bit: MyPlugin.a
- SUN 32 or 64-bit, Linux 32 or 64 bit, HP 32 or 64 bit on IPF: MyPlugin.so
- HP-UX 32 or 64-bit on PA-RISC: MyPlugin.sl

Note: The suffix "64" is only required on the library name for 64-bit Windows security plug-ins.

When you specify the name of a security plug-in in the database manager configuration, remember to use the full name of the library without the "64" suffix

and to omit both the file extension and any qualified path portion of the name. Regardless of the operating system, the security plug-in library called `MyPlugin` would be registered as follows:

```
UPDATE DBM CFG USING CLNT_PW_PLUGIN MyPlugin
```

The security plug-in name is case sensitive, and should exactly match the library name. DB2® will use the value from the database manager configuration parameter to assemble the library path, and then use the library path to load the security plug-in library.

To avoid security plug-in name conflicts, it is recommended that you name the plug-in based on the authentication method used, and an identifying symbol of the firm that wrote the plug-in. For instance, if the company `Foo, Inc.` wrote a plug-in implementing the authentication method `somemethod`, then the plug-in could have a name like `FOOsomemethod.DLL`.

The maximum length of a plug-in name (not including the file extension and the ″64″ suffix) is limited to 32 bytes. There is no maximum number of plug-ins supported by the database server, but the maximum length of the comma-separated list of plug-ins in the database manager configuration will be 255 bytes. There are two defines located in the include file `sqlenv.h` that establish these two limits:

```
#define SQL_PLUGIN_NAME_SZ     32     /* plug-in name */
#define SQL_SRVCON_GSSPLUGIN_LIST_SZ 255 /* GSS API plug-in list */
```

The security plug-in library files must have the following file permissions:

- Owned by the instance owner.
- Readable by all users on the system.
- Executable by all users on the system.

Related concepts:

- “Configuration parameters” in the *Administration Guide: Performance*
- “Security plug-ins” on page 533
- “Security plug-in library locations” on page 536

Related tasks:

- “Configuring DB2 with configuration parameters” in the *Administration Guide: Performance*
- “Deploying a group retrieval plug-in” on page 543
- “Deploying a user ID/password plug-in” on page 543
- “Deploying a GSS-API plug-in” on page 545
- “Deploying a Kerberos plug-in” on page 547

Related reference:

- “UPDATE DATABASE MANAGER CONFIGURATION Command” in the *Command Reference*
- “clnt_krb_plugin - Client Kerberos plug-in configuration parameter” in the *Administration Guide: Performance*
- “clnt_pw_plugin - Client userid-password plug-in configuration parameter” in the *Administration Guide: Performance*

Security plug-in support for two-part user IDs

The DB2® UDB for Linux, UNIX®, and Windows® product supports the use of two-part user IDs and the mapping of two-part user IDs to two-part authorization IDs.

For example, consider a Windows operating system two-part user ID composed of a domain and user ID such as: `MEDWAY\peter`. In this example of a two-part user ID `MEDWAY` is a domain and `peter` is the user name. In DB2, you can specify whether you this two-part user ID should be mapped to either a one-part authid or a two-part authid.

In DB2, prior to Version 8.2, you could only have a one-part user ID that mapped to a one-part authid. In DB2 Version 8.2, by default, one-part user IDs map to one-part authids and two-part user IDs map to one-part authids. The mapping of a two-part user ID to a two-part authid is supported, but is not the default behavior.

This allows a user to connect to the database using:

```
db2 connect to db user MEDWAY\peter using pw
```

In this case, if the default behavior is used, the user ID `MEDWAY\peter` will map to the authorization ID `PETER`. If the support for mapping a two-part user ID to a two-part authorization ID is used, the authorization ID in this case could be `MEDWAY\PETER`.

To enable DB2 support for authentication of two-part user IDs that map to two-part authids, you must enable the security plug-ins that perform the mapping of a two-part user ID to a two-part authid by setting database manager configuration parameters. These configuration parameters are discussed below.

DB2 supplies two sets of authentication plug-ins. One set is exclusively for mapping user IDs to one-part authids; that is for mapping a one-part user ID to a one-part authid and mapping a two-part user ID to a one-part authid. The second set adds the flexibly to map a one-part user ID to a one-part authid and maps a two-part user ID to a two-part authid.

If a user name in your work environment can be mapped to multiple accounts defined in different locations (such as local account, domain account, and trusted domain accounts), then you may want to specify the plug-ins that enable the two-part authid mapping.

It is important to note that a one-part authid, such as, `PETER` and a two-part authid that combines a domain and a user ID like `MEDWAY\PETER` are functionally distinct authids. The set of privileges associated with one of these authids is completely distinct from the set of privileges associated with the other authid. Care should be taken when working with one-part and two-part authids.

The following table identifies the kinds of plug-ins supplied by DB2, and the plug-in names for the specific authentication implementations.

Table 82. DB2 security plug-ins

Authentication type	Name of one-part user ID plug-in	Name of two-part user ID plug-in
User ID/password (Client)	IBMOSauthclient	IBMOSauthclientTwoPart

Table 82. DB2 security plug-ins (continued)

Authentication type	Name of one-part user ID plug-in	Name of two-part user ID plug-in
User ID/password (Server)	IBMOSauthserver	IBMOSauthserverTwoPart
Kerberos	IBMkrb5	IBMkrb5TwoPart

Note: On Windows 64-bit platforms, the characters ″64″ are appended to the plug-in names listed here.

To map a two-part user ID to a two-part authid, you must specify that the two-part plug-in, which is the non-default plug-in, be used. Security plug-ins are specified at the instance level by setting the security related database manager configuration parameters as follows:

For server authentication that maps two-part user IDs to two-part authids, you must set:

- `SRVCON_PW_PLUGIN` to `IBMOSauthserverTwoPart`
- `CLNT_PW_PLUGIN` to `IBMOSauthclientTwoPart`

For client authentication that maps two-part user IDs to two-part authids, you must set:

- `SRVCON_PW_PLUGIN` to `IBMOSauthserverTwoPart`
- `CLNT_PW_PLUGIN` to `IBMOSauthclientTwoPart`

For Kerberos authentication that maps two-part user IDs to two-part authids, you must set:

- `SRVCON_GSSPLUGIN_LIST` to `IBMOSkrb5TwoPart`
- `CLNT_KRB_PLUGIN` to `IBMkrb5TwoPart`

The security plug-in libraries accept two-part user IDs specified in a Microsoft® Windows Security Account Manager compatible format. For example, in the format: domain\user ID. Both the domain and user ID information will be used by the DB2 authentication and authorization processes at connection time.

When you specify an authentication type that requires a user ID/password or Kerberos plug-in, the plug-ins that are listed in the ″Name of one-part user ID plug-in″ column are used by default.

It is advised that the two-part plug-ins be considered for implementation when creating new databases to avoid conflicts with one-part authids in existing databases. New databases that will make use of the two-part authid authentication must be created in a separate instance than databases that use single-part authids.

Related concepts:

- “DB2 for Windows NT and Windows NT security introduction” in the *Administration Guide: Implementation*

Related tasks:

- “DB2 for Windows NT authentication with groups and domain security” in the *Administration Guide: Implementation*

Related reference:

- "clnt_pw_plugin - Client userid-password plug-in configuration parameter" in the *Administration Guide: Performance*
- "srvcon_pw_plugin - Userid-password plug-in for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*

32-bit and 64-bit considerations for security plug-ins

In general, a 32-bit DB2® instance will use the 32-bit security plug-in and 64-bit DB2 instance will use the 64-bit security plug-in. However, on a 64-bit instance, DB2 supports 32-bit applications, which will require the 32-bit plug-in library.

A database instance where both the 32-bit and the 64-bit applications can run is known as a hybrid instance. If you have a hybrid instance and intend to run 32-bit applications, ensure that the required 32-bit security plug-ins are available in the 32-bit plug-in directory. For hybrid DB2 instances on a UNIX® operating system, the directories `security32` and `security64` appear. For a Windows® 64-bit hybrid instance, both 32-bit and 64-bit security plug-ins are located in the same directory, but 64-bit plug-in names have a suffix, "64".

If you wish to migrate from a 32-bit instance to a 64-bit instance, you should obtain versions of your security plug-ins that are recompiled for 64-bit.

If you acquired your security plug-ins from a vendor who will not supply 64-bit plug-in libraries, you can implement a 64-bit stub that executes a 32-bit application. In this situation, the security plug-in is an external program rather than a library.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Related tasks:

- "Migrating applications from 32-bit to 64-bit environments" in the *Application Development Guide: Building and Running Applications*

Security plug-in problem determination

Problems with security plug-ins are reported in two ways: through SQL errors and through the administrative log.

Following are the SQLCODE values related to security plug-ins:

- SQLCODE -1365 is returned when a plug-in error occurs during db2start or db2stop.
- SQLCODE -1366 is returned whenever there is a local authorization problem.
- SQLCODE -30082 is returned for all connection-related plug-in errors.

The administrative log is a good resource for debugging and administrating security plug-ins. To see the administrative log on UNIX®, check `sqllib/db2dump/<instance name>.nfy`. To see the administrative log on Windows operating systems, use the Event Viewer tool. The Event Viewer tool can be found by navigating from the Windows operating system "Start" button to `Settings -> Control Panel -> Administrative Tools -l> Event Viewer`. Following are the administration log values related to security plug-ins:

- 13000 indicates that a call to a GSS-API security plug-in API failed with an error, and returned an optional error message.

  ```
  SQLT_ADMIN_GSS_API_ERROR (13000)
  Plug-in "<plug-in name>" received error code "<error code>" from
  GSS API  "<gss api name>" with the error message "<error message>"
  ```

- 13001 indicates that a call to a DB2® security plug-in API failed with an error, and returned an optional error message.

  ```
  SQLT_ADMIN_PLUGIN_API_ERROR(13001)
  Plug-in "<plug-in name>" received error code "<error code>" from DB2
  security plug-in API "<gss api name>" with the error message
  "<error message>"
  ```

- 13002 indicates that DB2 failed to unload a plug-in.

  ```
  SQLT_ADMIN_PLUGIN_UNLOAD_ERROR (13002)
  Unable to unload plug-in "<plug-in name>". No further action required.
  ```

- 13003 indicates a bad principal name.

  ```
  SQLT_ADMIN_INVALID_PRIN_NAME (13003)
  The principal name "<principal name>" used for "<plug-in name>"
  is invalid. Fix the principal name.
  ```

- 13004 indicates that the plug-in name is not valid. Path separators (On UNIX ″/″ and on Windows® ″\″) are not allowed to be used as part of the plug-in name.

  ```
  SQLT_ADMIN_INVALID_PLGN_NAME (13004)
  The plug-in name "<plug-in name>" is invalid. Fix the plug-in name.
  ```

- 13005 indicates that the security plug-in failed to load. Ensure the plug-in is in the correct directory and that the appropriate database manager configuration parameters are updated.

  ```
  SQLT_ADMIN_PLUGIN_LOAD_ERROR (13005)
  Unable to load plug-in "<plug-in name>". Verify the plug-in existence and
  directory where it is located is correct.
  ```

- 13006 indicates that an unexpected error was encountered by a security plug-in. Gather all the db2support information, if possible capture a db2trc, and then call IBM® support for further assistance.

  ```
  SQLT_ADMIN_PLUGIN_UNEXP_ERROR (13006)
  Plug-in encountered unexpected error. Contact IBM Support for further assistance.
  ```

Note: If you are using security plug-ins on a Windows 64-bit database server and are seeing a load error for a security plug-in, consult the topics 32-bit and 64-bit considerations for security plug-ins and Security plug-in naming conventions. The 64-bit plug-in library requires the suffix ″64″ on the library name, but the entry in the security plug-in database manager configuration parameters should not indicate this suffix.

Related concepts:

- “Event monitors” in the *System Monitor Guide and Reference*
- “Error Information in the SQLCODE, SQLSTATE, and SQLWARN Fields” on page 100
- “SQLSTATE and SQLCODE Variables in C and C++” on page 168
- “SQLCODE and SQLSTATE Differences among IBM Relational Database Systems” on page 697
- “DB2 trace (db2trc)” in the *Troubleshooting Guide*
- “Security plug-ins” on page 533
- “32-bit and 64-bit considerations for security plug-ins” on page 541
- “Security plug-in APIs” on page 559

Related reference:

- "CREATE EVENT MONITOR statement" in the *SQL Reference, Volume 2*
- "db2trc - Trace Command" in the *Command Reference*
- "Error messages for security plug-ins" on page 554
- "Required APIs and Definitions for GSS-API authentication plug-ins" on page 591
- "APIs for group retrieval plug-ins" on page 560
- "APIs for user ID/password authentication plug-in" on page 569

Deploying a group retrieval plug-in

If you want to customize the DB2 security system's group retrieval behavior, you can develop your own group retrieval plug-in or buy one from a third party.

Once you acquire a group retrieval plug-in that is suitable for your database management system, you can deploy it.

Procedure:

To deploy a group retrieval plug-in on the database server, perform the following steps:

1. Place the group retrieval plug-in library in the server's group plug-in directory.
2. Update the database manager configuration parameter *group_plugin* with the name of the plug-in.

To deploy a group retrieval plug-in on database clients, perform the following steps:

1. Place the group retrieval plug-in library in the client's group plug-in directory.
2. Update the database manager configuration parameter *group_plugin* with the name of the plug-in.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536
- "Security plug-in naming conventions" on page 537

Related tasks:

- "Deploying a user ID/password plug-in" on page 543
- "Deploying a GSS-API plug-in" on page 545
- "Deploying a Kerberos plug-in" on page 547

Related reference:

- "group_plugin - Group plug-in configuration parameter" in the *Administration Guide: Performance*

Deploying a user ID/password plug-in

If you want to customize the DB2 security system's user ID/password authentication behavior, you can develop your own user ID/password authentication plug-ins or buy one from a third party.

Once you acquire user ID/password authentication plug-ins that are suitable for your database management system, you can deploy them.

All user ID-password based authentication plug-ins must be placed in either the client plug-in directory or the server plug-in directory depending on the intended usage of the plug-ins. If a plug-in is placed in the client plug-in directory, it will be used for local authorization checking and whenever a client attempts to connect with the server. If the plug-in is placed in the server plug-in directory, it will be used for handling incoming connections to the server and for checking whether an authid exists and is valid whenever the GRANT statement is issued without specifying either the keyword USER or GROUP.

In most cases user ID/password authentication requires only a server-side plug-in. It is possible, though generally deemed less useful, to have only a client user ID/password plug-in. It is possible though quite unusual to require matching user ID/password plug-ins on both the client and the server.

Procedure:

To deploy a user ID/password authentication plug-in on the database server, perform the following steps:

1. Place the user ID/password authentication plug-in library in the server's plug-in directory.
2. Update the database manager configuration parameter *srvcon_pw_plugin* with the name of the server plug-in.

   This plug-in is used by the server when it is handling connection (CONNECT) and attachment (ATTACH) requests.
3. Either:
   - Set the database manager configuration parameter *srvcon_auth* to the CLIENT, SERVER, SERVER_ENCRYPT, DATA_ENCRYPT, or DATA_ENCRYPT_CMP authentication type. Or:
   - Set the database manager configuration parameter *srvcon_auth* to NOT_SPECIFIED and set *authentication* to CLIENT, SERVER, SERVER_ENCRYPT, DATA_ENCRYPT, or DATA_ENCRYPT_CMP authentication type.

To deploy a user ID/password authentication plug-in on database clients, perform the following steps:

1. Place the user ID/password authentication plug-in library in the client plug-in directory on the client.
2. Update the database manager configuration parameter *clnt_pw_plugin* with the name of the client plug-in.

   This plug-in is loaded and called regardless of where the authentication is being done, that is, not only when client authentication is enabled.

For local authorization on a client, server, or gateway, using a user ID/password authentication plug-in, perform the following steps:

1. Place the user ID/password authentication plug-in library in the client plug-in directory on the client, server, or gateway.
2. Update the database manager configuration parameter *clnt_pw_plugin* with the name of the plug-in.
3. Set the *authentication* database manager configuration parameter to CLIENT, SERVER, SERVER_ENCRYPT, DATA_ENCRYPT, or DATA_ENCRYPT_CMP.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Related tasks:

- "Deploying a group retrieval plug-in" on page 543
- "Deploying a GSS-API plug-in" on page 545

Related reference:

- "authentication - Authentication type configuration parameter" in the *Administration Guide: Performance*
- "GRANT (Database Authorities) statement" in the *SQL Reference, Volume 2*
- "clnt_pw_plugin - Client userid-password plug-in configuration parameter" in the *Administration Guide: Performance*
- "srvcon_auth - Authentication type for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*
- "srvcon_pw_plugin - Userid-password plug-in for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*

Deploying a GSS-API plug-in

If you want to customize the DB2 security system's authentication behavior, you can develop your own authentication plug-ins using the GSS-API, or buy one from a third party.

Once you acquire GSS-API authentication plug-ins that are suitable for your database management system, you can deploy them.

In the case of a GSS-API or Kerberos plug-ins, you must have matching plug-in types on the client and the server. The plug-ins on the client and server need not be from the same vendor, but they must generate and consume compatible GSS-API tokens. For example, any combination of Kerberos plug-ins deployed on the client and the server is okay since Kerberos plug-ins are standardized, however, different implementations of less standardized GSS-API mechanisms, such as *x.509* certificates might not be completely compatible.

All GSS-API authentication plug-ins must be placed in either the client plug-in directory or the server plug-in directory depending on the intended usage of the plug-ins. If a plug-in is placed in the client plug-in directory, it will be used for local authorization checking and when a client attempts to connect with the server. If the plug-in is placed in the server plug-in directory, it will be used for handling incoming connections to the server and for checking whether an authid exists and is valid whenever the GRANT statement is issued without specifying either the keyword USER or GROUP.

Procedure:

To deploy a GSS-API authentication plug-in on the database server, perform the following steps:

1. Place the GSS-API authentication plug-in library in the server plug-in directory on the server. You can copy numerous GSS-API plug-ins into this directory.

2. Update the database manager configuration parameter *srvcon_gssplugin_list* with an ordered, comma-delimited list of the names of the plug-ins installed in the GSS-API plug-in directory.
3. Either:
   - Set the database manager configuration parameter *srvcon_auth* to GSSPLUGIN. Or:
   - Set the database manager configuration parameter *srvcon_auth* to NOT_SPECIFIED and set *authentication* to GSSPLUGIN.

To deploy a GSS-API authentication plug-in on database clients, perform the following steps:

1. Place the GSS-API authentication plug-in library in the client plug-in directory on the client. You can copy numerous GSS-API plug-ins into this directory. The client selects the appropriate GSS-API plug-in for authentication during CONNECT/ATTACH by picking the first GSS-API plug-in contained in the server's plug-in list on the client.
2. Optional: Catalog the databases that the client will access, indicating that the client will only accept a GSS-API authentication plug-in as the authentication mechanism. For example:

   ```
   CATALOG DB testdb AT NODE testnode AUTHENTICATION GSSPLUGIN
   ```

For local authorization on a client, server, or gateway, using a GSS-API authentication plug-in, perform the following steps:

1. Place the GSS-API authentication plug-in library in the client plug-in directory on the client, server, or gateway.
2. Update the database manager configuration parameter *local_gssplugin* with the name of the plug-in.
3. Set the *authentication* database manager configuration parameter to GSSPLUGIN, or GSS_SERVER_ENCRYPT.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Related reference:

- "authentication - Authentication type configuration parameter" in the *Administration Guide: Performance*
- "CATALOG DATABASE Command" in the *Command Reference*
- "local_gssplugin - GSS API plug-in used for local instance level authorization configuration parameter" in the *Administration Guide: Performance*
- "srvcon_auth - Authentication type for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*
- "srvcon_gssplugin_list - List of GSS API plug-ins for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*
- "Security plug-in samples" in the *Application Development Guide: Building and Running Applications*

Deploying a Kerberos plug-in

If you want to customize the DB2 security system's Kerberos authentication behavior, you can develop your own Kerberos authentication plug-ins or buy one from a third party.

Once you acquire Kerberos authentication plug-ins that are suitable for your database management system, you can deploy them.

Procedure:

To deploy a Kerberos authentication plug-in on the database server, perform the following steps:

1. Place the Kerberos authentication plug-in library in the server plug-in directory on the server.
2. Update the database manager configuration parameter *srvcon_gssplugin_list*, which is presented as an ordered, comma delimited list, to include the Kerberos server plug-in name. Only one plug-in in this list can be a Kerberos plug-in.

   If this list is blank and *authentication* is set to KERBEROS or KRB_SVR_ENCRYPT, the default DB2 Kerberos plug-in: IBMkrb5 will be used.
3. Either:
   - Set the database manager configuration parameter *srvcon_auth* to the KERBEROS or KRB_SERVER_ENCRYPT authentication type. Or:
   - Set the database manager configuration parameter *srvcon_auth* to NOT_SPECIFIED and set *authentication* to KERBEROS or KRB_SERVER_ENCRYPT authentication type.

To deploy a Kerberos authentication plug-in on database clients, perform the following steps:

1. Place the Kerberos authentication plug-in library in the client plug-in directory on the client.
2. Update the database manager configuration parameter *clnt_krb_plugin* with the name of the client Kerberos plug-in.

   If *clnt_krb_plugin* is blank, DB2 assumes that the client cannot use Kerberos authentication. This is only appropriate when the server cannot support plug-ins. See the limititaions on the use of security plug-ins for more information. If both the server and the client support security plug-ins, then the client will not use the value of *clnt_krb_plugin* because the server has a GSS-API plug-in with the name *IBMkrb5* listed.

   For local authorization on a client, server, or gateway, using a Kerberos authentication plug-in, perform the following steps:

   a. Place the Kerberos authentication plug-in library in the client plug-in directory on the client, server, or gateway.
   b. Update the database manager configuration parameter *clnt_krb_plugin* with the name of the plug-in.
   c. Set the *authentication* database manager configuration parameter to KERBEROS, or KRB_SERVER_ENCRYPT.

   The Kerberos plug-in provided by DB2 is named `IBMkrb5`.
3. Optional: Catalog the databases that the client will access, indicating that the client will only use a Kerberos authentication plug-in. For example:

```
CATALOG DB testdb AT NODE testnode AUTHENTICATION KERBEROS
        TARGET PRINCIPAL service/host@REALM
```

Note: For platforms supporting Kerberos, the `IBMkrb5` library will be present in the client plug-in directory. DB2 will recognize this library as a valid GSS-API plug-in, because Kerberos plug-ins are implemented using GSS-API. plug-in.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Related tasks:

- "Deploying a group retrieval plug-in" on page 543
- "Deploying a user ID/password plug-in" on page 543
- "Deploying a GSS-API plug-in" on page 545

Related reference:

- "authentication - Authentication type configuration parameter" in the *Administration Guide: Performance*
- "CATALOG DATABASE Command" in the *Command Reference*
- "clnt_krb_plugin - Client Kerberos plug-in configuration parameter" in the *Administration Guide: Performance*
- "srvcon_auth - Authentication type for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*
- "srvcon_gssplugin_list - List of GSS API plug-ins for incoming connections at the server configuration parameter" in the *Administration Guide: Performance*

Chapter 27. Developing security plug-ins

How DB2 loads security plug-ins

Each plug-in library must contain an initialization function with a specific name determined by the plug-in type:

- Server side authentication plug-in: db2secServerAuthPluginInit()
- Client side authentication plug-in: db2secClientAuthPluginInit()
- Group plug-in: db2secGroupPluginInit()

This function is known as the plug-in initialization function. The plug-in initialization function initializes the specified plug-in and provides DB2® with information that it requires to call the plug-in's functions. The plug-in initialization function accepts the following parameters:

- the highest version number of the functions pointer structure that DB2 can support
- pointer to a structure containing pointers to all the APIs requiring implementation
- pointer to a function that adds log messages to the db2diag.log file
- pointer to an error message string
- length of the error message

The following is a function signature for the initialization function of a group retrieval plug-in:

```
SQL_API_RC SQL_API_FN db2secGroupPluginInit(
 db2int32 version,
 void *group_fns,
  db2secLogMessage *logMessage_fn,
  char **errormsg,
 db2int32  *errormsglen);
```

Note: Plug-in libraries can only be implemented in C or C++. If the plug-in library is compiled as C++, all functions must be declared with: `extern "C"`. DB2 relies on the underlying operating system dynamic loader to handle the C++ constructors and destructors used inside of a C++ user-written plug-in library.

This is the only function in the plug-in library that must have a prescribed function name. The other plug-in functions are referenced through function pointers returned from the initialization function. Server plug-ins are loaded at db2start time on the server. Client plug-ins are loaded when required on the client. Immediately after DB2 loads the plug-in library, it will resolve the location of this function and then call it. The specific task of this function is as follows:

- cast the functions pointer to a pointer to an appropriate functions structure
- fill in the pointers to the other functions in the library
- fill in the version number of the function pointer structure being returned

DB2 can potentially call the plug-in initialization function more than once. A situation where this is would occur is when an application dynamically loads the DB2 client library, unloads it and reloads it again, and then performs authentication functions from a plug-in both before and after reloading. In this case, the plug-in library might not be unloaded and then re-loaded, however, this behavior varies depending on the operating system.

Another example of DB2 issuing multiple calls to a plug-in initialization function is during the execution of stored procedures or federated system calls, where the database server can itself act as a client. If the client and server plug-ins on the database server are in the same file, DB2 could call the plug-in initialization function twice.

If the plug-in detects that `db2secGroupPluginInit` is called more than once, it should handle this as if it was directed to terminate and reinitialize the plug-in library. As such, the plug-in initialization function should do the entire cleanup that a call to `db2secPluginTerm` would do before returning the set of function pointers again.

On a DB2 server running a UNIX® operating system, DB2 can potentially load and initialize plug-in libraries more than once after db2start in different processes.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Related reference:

- "Restrictions on security plug-in libraries" on page 550
- "Return codes for security plug-ins" on page 552
- "Calling sequences for the security plug-in APIs" on page 555
- "db2secGroupPluginInit - Initialize group plug-in" on page 562
- "db2secPluginTerm - Clean up group plug-in resources" on page 563
- "db2secClientAuthPluginInit - Initialize client authentication plug-in" on page 576
- "db2secServerAuthPluginInit - Initialize server authentication plug-in" on page 587

Restrictions on security plug-in libraries

Following are restrictions for developing plug-in libraries.

C-linkage
Plug-in libraries must be linked with C-linkage. Header files providing the prototypes, data structures needed to implement the plug-ins, and error code definitions will be provided for C/C++ only. Functions that DB2 will resolve at load time must be declared with extern ″C″ if the plug-in library is compiled as C++.

.NET common language runtime is not supported
The .NET common language runtime (CLR) is not supported for compiling and linking source code for plug-in libraries.

Signal handlers
The plug-in libraries must not install signal handlers or change the signal mask, since doing so will interfere with DB2's signal handlers. Interfering

with the DB2 signal handlers could seriously interfere with DB2's ability to report and recover from errors, including traps in the plug-in code itself. Plug-in libraries should also never throw C++ exceptions, as this can also interfere with DB2's error handling.

Thread-safe
Plug-in libraries must be thread-safe and re-entrant. The plug-in initialization function is the only API that is not required to be re-entrant. This is because the plug-in initialization function could potentially be called multiple times from different processes; in which case, the plug-in will cleanup all used resources and reinitialize itself.

Exit handlers and overriding standard C library and operating system calls
Plug-in libraries should not override standard C library or operating system calls. Plug-in libraries should also not install at exit handlers or `pthread_atfork` handlers. The use of exit handlers is not recommended because they may be unloaded before the program exits.

Library dependencies
On Linux or Unix the processes that load the plug-in libraries can be setuid or setgid, which means that they will not be able to rely n the `$LD_LIBRARY_PATH`, `$SHLIB_PATH`, or `$LIBPATH` environment variables to find dependent libraries. Therefore, plug-in libraries should not depend on other libraries, unless any dependant libraries are accessible through other methods, such as the following:

- by being in `/lib` or `/usr/lib`
- by having the directories they reside in being specified OS-wide (such as in the `ld.so.conf` file on Linux)
- by being specified in the RPATH in the plug-in library itself

This restriction is not applicable to Windows operating systems.

Symbol collisions
When possible, plug-in libraries should be compiled and linked with any available options that reduce the likelihood of symbol collisions, such as those that reduce unbound external symbolic references. For example, use of the "-Bsymbolic" linker option on HP, Sun Solaris, and Linux can help prevent problems related to symbol collisions. However, for a plug-in written on AIX platform, please do not use `"-brtl"` linker option explicitly or implicitly, because this will cause a problem.

32-bit and 64-bit applications
32-bit applications must use 32-bit plug-ins. 64-bit applications must use 64-bit plug-ins. Please refer to the topic 32-bit and 64-bit considerations for security plug-ins for more details.

Text strings
Input text strings are not guaranteed to be null-terminated, and output strings are not required to be null-terminated. Instead, integer lengths are given for all input strings, and pointers to integers are given for lengths to be returned.

Passing authid parameters
An authid parameter that DB2 passes into a plug-in (an input authid parameter) will contain an upper-case authid, with padded blanks removed. An authid parameter that a plug-in returns to DB2 (an output authid parameter) does not require any special treatment, but DB2 will take the authid and make it upper-case, and padded with blanks according to the internal DB2 standard.

Size limits for parameters

The plug-in APIs use the following as length limits for parameters:

```
#define DB2SEC_MAX_AUTHID_LENGTH 255
#define DB2SEC_MAX_USERID_LENGTH 255
#define DB2SEC_MAX_USERNAMESPACE_LENGTH 255
#define DB2SEC_MAX_PASSWORD_LENGTH 255
#define DB2SEC_MAX_DBNAME_LENGTH 128
```

A particular plug-in implementation may require or enforce smaller maximum lengths for the authorization IDs, user IDs, and passwords. In particular, the operating system authentication plug-ins supplied with DB2 UDB are restricted to the maximum user, group and namespace length limits enforced by the operating system for cases where the operating system limits are lower than those stated above.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in library locations" on page 536

Return codes for security plug-ins

All security plug-in APIs must return an integer value for the purpose of indicating success or failure of the execution of the API. A return code value of 0 indicates that the API ran successfully. All negative return codes, with the exception of -3, -4, and -5, indicate that the API encountered an error.

All negative return codes returned from the security-plug-in APIS will be mapped to SQLCODE -1365, SQLCODE -1366, or SQLCODE -30082, with the exception of return codes with the -3, -4, or -5. The values -3, -4, and -5 are used to indicate whether or not an AUTHID is represent a valid user or group.

All the security plug-in API return codes are defined in `db2secPlugin.h`, which can be found in DB2's include directory: `SQLLIB/include`.

Details regarding all of the security plug-in return codes are presented in the following table:

Table 83. Security plug-in return codes

Return code	Define value	Meaning	Applicable APIs
0	DB2SEC_PLUGIN_OK	The plug-in API executed successfully.	All
-1	DB2SEC_PLUGIN _UNKNOWNERROR	The plug-in API encountered an unexpected error.	All
-2	DB2SEC_PLUGIN_BADUSER	The user ID passed in as input is not defined.	db2secGenerateInitialCred db2secValidatePassword db2secRemapUserid db2secGetGroupsForUser
-3	DB2SEC_PLUGIN _INVALIDUSERORGROUP	No such user or group. /entry>	db2secDoesAuthIDExist db2secDoesGroupExist
-4	DB2SEC_PLUGIN _USERSTATUSNOTKNOWN	Unknown user status. This is not treated as an error by DB2; it is used by a GRANT statement to determine if an authid represent an user or an operating system group.	db2secDoesAuthIDExist

Table 83. Security plug-in return codes (continued)

Return code	Define value	Meaning	Applicable APIs
-5	DB2SEC_PLUGIN _GROUPSTATUSNOTKNOWN	Unknown group status. This is not treated as an error by DB2; it is used by a GRANT statement to determine if an authid represent an user or an operating system group.	db2secDoesGroupExist
-6	DB2SEC_PLUGIN_UID_EXPIRED	Userid expired.	db2secValidatePassword db2GetGroupsForUser db2secGenerateInitialCred
-7	DB2SEC_PLUGIN_PWD_EXPIRED	Password expired.	db2secValidatePassword db2GetGroupsForUser db2secGenerateInitialCred
-8	DB2SEC_PLUGIN_USER_REVOKED	User revoked.	db2secValidatePassword db2GetGroupsForUser
-9	DB2SEC_PLUGIN _USER_SUSPENDED	User suspended.	db2secValidatePassword db2GetGroupsForUser
-10	DB2SEC_PLUGIN_BADPWD	Bad password.	db2secValidatePassword db2secRemapUserid db2secGenerateInitialCred
-11	DB2SEC_PLUGIN _BAD_NEWPASSWORD	Bad new password.	db2secValidatePassword db2secRemapUserid
-12	DB2SEC_PLUGIN _CHANGEPASSWORD _NOTSUPPORTED	Change password not supported.	db2secValidatePassword db2secRemapUserid db2secGenerateInitialCred
-13	DB2SEC_PLUGIN_NOMEM	Plug-in attempt to allocate memory failed due to insufficient memory.	All
-14	DB2SEC_PLUGIN_DISKERROR	Plug-in encountered a disk error.	All
-15	DB2SEC_PLUGIN_NOPERM	Plug-in attempt to access a file failed because of wrong permissions on the file.	All
-16	DB2SEC_PLUGIN_NETWORKERROR	Plug-in encountered a network error.	All
-17	DB2SEC_PLUGIN _CANTLOADLIBRARY	Plug-in is unable to load a required library.	db2secGroupPluginInit db2secClientAuthPluginInit db2secServerAuthPluginInit
-18	DB2SEC_PLUGIN_CANT _OPEN_FILE	Plug-in is unable to open and read a file for a reason other than missing file or inadequate file permissions.	All
-19	DB2SEC_PLUGIN_FILENOTFOUND	Plug-in is unable to open and read a file, because the file is missing from the file system.	All
-20	DB2SEC_PLUGIN _CONNECTION_DISALLOWED	The plug-in is refusing the connection because of the restriction on which database is allowed to connect, or which TCP/IP address can connect to a specific database.	All server-side plug-in APIs.
-21	DB2SEC_PLUGIN_NO_CRED	GSS API plug-in only: Initial client credential is missing.	db2secGetDefaultLoginContext db2secServerAuthPluginInit
-22	DB2SEC_PLUGIN_CRED_EXPIRED	GSS API plug-in only: Client credential has expired.	db2secGetDefaultLoginContext db2secServerAuthPluginInit
-23	DB2SEC_PLUGIN _BAD_PRINCIPAL_NAME	GSS API plug-in only: The principal name is invalid.	db2secProcessServerPrincipalName
-24	DB2SEC_PLUGIN _NO_CON_DETAILS	This return code is returned by the `db2secGetConDetails` callback (for example, from DB2 to the plug-in) to indicate that DB2 is unable to determine the client's TCP/IP address.	db2secGetConDetails

Table 83. Security plug-in return codes (continued)

Return code	Define value	Meaning	Applicable APIs
-25	DB2SEC_PLUGIN _BAD_INPUT_PARAMETERS	Some parameters are not valid or are missing when plug-in API is called.	All
-26	DB2SEC_PLUGIN _INCOMPATIBLE_VER	The version of the APIs reported by the plug-in is not compatible with DB2.	db2secGroupPluginInit db2secClientAuthPluginInit db2secServerAuthPluginInit
-27	DB2SEC_PLUGIN_PROCESS_LIMIT	The plug-in ran out of resources when attempting to create a new process.	All
-28	DB2SEC_PLUGIN_NO_LICENSES	The plug-in encountered a user license problem. A possibility exists that the underlying mechanism license has reached the limit.	All

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in problem determination" on page 541

Error messages for security plug-ins

When an error occurs in a security plug-in API, the API can return an ASCII text string in the `errormsg` field to provide a more specific description of the problem than the return code. For instance, the `errormsg` string can contain `"File /home/db2inst1/mypasswd.txt does not exist."` DB2 will write this entire string into the DB2 administration notification log, and will also include a truncated version as a token in some SQL messages. Since tokens in SQL messages can only be of limited length, it is therefore recommended that these messages be kept short, and that important variable portions of these messages appear at the front of the string. To aid in debugging, consider adding the name of the security plug-in to the error message.

For non-urgent errors, such as password expired errors, the `errormsg` string will only be dumped when the DIAGLEVEL database manager configuration parameter is set at 4.

The memory for these error messages must be allocated by the security plug-in. The plug-ins must therefore also provide an API to free this memory: `db2secFreeErrormsg`.

The `errormsg` field will only be checked by DB2 if an API returns a non-zero value. The plug-in should therefore not allocate memory for this returned error message if there is no error.

At initialization time a message logging function pointer, `logMessage_fn`, is passed to the group, client and server plug-ins. The plug-ins can use the function to log any debugging information to db2diag.log. For example:

```
// Log an message indicate init successful
(*(logMessage_fn))(DB2SEC_LOG_CRITICAL,
                   "db2secGroupPluginInit successful",
                    strlen("db2secGroupPluginInit successful"));
```

For more details about each parameter for the db2secLogMessage function, please refer to the initialization API for each of the plug-in types.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in problem determination" on page 541
- "Security plug-in APIs" on page 559
- "Security plug-in support for two-part user IDs" on page 539

Related reference:

- "Return codes for security plug-ins" on page 552

Calling sequences for the security plug-in APIs

There are five main scenarios in which DB2 will call security plug-in APIs:

- On a client for a database connection.
- On a client, server, or gateway for local authorization.
- On a server for a database connection.
- On a server for a grant statement.
- On a server to get a list of groups that an authid belongs to.

Note: The DB2 server treats database actions requiring local authorizations, such as db2start, db2stop, and db2trc, like client applications.

For each of these operations, the sequence with which DB2 calls security plug-in APIs is appropriately different. Following are the sequences of APIs called by DB2 for each of these scenarios.

On a client for a database connection

When the user-configured authentication type is CLIENT, the DB2 client application will call the following security plug-in APIs:

- `db2secGetDefaultLoginContext();`
- `db2secValidatePassword();`
- `db2secFreetoken();`

In the case of an implicit authentication, that is, when you connect without specifying a particular user ID or password, the `db2secValidatePassword` API will be called if you are using a user ID/password plug-in. This API permits plug-in developers to prohibit implicit authentication if necessary.

On an implicit authentication, if the database manager configuration parameter, *authentication*, is set to anything other than CLIENT (implying authentication at the server), the application will call the following security plug-in APIs for the user ID/password authentication mechanism:

- `db2secGetDefaultLoginContext();`
- `db2secFreeToken();`

On an implicit authentication, if *authentication* is set to anything other than CLIENT (implying authentication at the server), the application will call the following security plug-in APIs for GSS-API plug-ins. (The call to `gss_init_sec_context()` will use GSS_C_NO_CREDENTIAL as the input credential.)

- `db2secGetDefaultLoginContext();`
- `db2secProcessServerPrincipalName();`
- `gss_init_sec_context();`

- gss_release_buffer();
- gss_release_name();
- gss_delete_sec_context();
- db2secFreeToken();

The API *gss_init_sec_context()* may be called twice if a mutual authentication token is returned from the server.

On an explicit authentication, if *authentication* is set to CLIENT the DB2 client application will call the following security plug-in APIs:

- db2secRemapUserid();
- db2secValidatePassword();
- db2secFreeToken();

On an explicit authentication, if *authentication* is set to anything other than CLIENT, the application will call the following security plug-in APIs for the user ID/password authentication mechanism:

- db2secRemapUserid();

If the negotiated authentication type is GSS-API or Kerberos, the client application will call the following security plug-in APIs for GSS-API plugins in the following ordered sequence. These APIs are used for implicit or explicit authentication (a connection to a database in which both the user ID and password are specified) unless otherwise stated.

- db2secProcessServerPrincipalName();
- db2secGenerateInitialCred(); (For explicit authentication only)
- gss_init_sec_context();
- gss_release_buffer ();
- gss_release_name();
- gss_release_cred();
- db2secFreeInitInfo();
- gss_delete_sec_context();
- db2secFreeToken();

The API gss_init_sec_context() may be called twice if a mutual authentication token is returned from the server.

On a client, server, or gateway for local authorization

For a local authorization, the DB2 command being used will call the following security plug-in APIs:

- db2secGetDefaultLoginContext();
- db2secGetGroupsForUser();
- db2secFreeToken();
- db2secFreeGroupList();

These APIs will be called for both user ID/password and GSS-API authentication mechanisms.

On a server for a database connection

For a database connection on the database server, the DB2 agent process or thread will call the following security plug-in APIs for the user ID/password authentication mechanism:

- `db2secValidatePassword();` Only if *authentication* is not CLIENT
- `db2secGetAuthIDs();`
- `db2secGetGroupsForUser();`
- `db2secFreeToken();`
- `db2secFreeGroupList();`

For a database connection on the database server, the DB2 agent process or thread will call the following security plug-in APIs for the GSS-API authentication mechanism:

- `gss_accept_sec_context();`
- `gss_release_buffer();`
- `db2secGetAuthIDs();`
- `db2secGetGroupsForUser();`
- `gss_delete_sec_context();`
- `db2secFreeToken();`
- `db2secFreeGroupList();`

On a server for a GRANT statement

For a GRANT statement that does not specify the USER or GROUP keyword, (for example, `"GRANT CONNECT ON DATABASE TO user1"`), DB2 must be able to determine if `user1` is a user, a group, or both. Therefore, DB2 will call the following security plug-in APIs:

- `db2secDoesGroupExist();`
- `db2secDoesAuthIDExist();`

On a server to get a list of groups to which an authid belongs

From your database server, when you need to get a list of groups to which an authid belongs, DB2 will call the following security plug-in API with only the authid as input:

- `db2secGetGroupsForUser();`

There will be no token from other security plug-ins.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Chapter 28. Security plug-in APIs

Security plug-in APIs

To enable you to customize DB2®'s authorization behavior, DB2 has exposed APIs that you can use to modify existing plug-ins or build new security plug-ins.

When you develop a security plug-in, you will need to implement the standard authentication functions that DB2 will invoke. For the three available types of plug-ins, the functionality you will need to implement is as follows:

Group retrieval
Retrieves group membership information for a given user and determine if a given string represents a valid group name.

User ID/password authentication
Authentication that identifies the default security context (client only), validate and optionally change a password, determine if a given string represents a valid user (server only), modify the user ID or password provided on the client before it is sent to the server (client only), return the DB2 authorization ID associated with a given user.

GSS-API authentication
Authentication that implements the required GSS-API functions, identify the default security context (client side only), generate initial credentials basedon user ID and password and optionally change password (client side only), create and accept security tickets, and return the DB2 authorization ID associated with a given GSS-API security context.

The following are definitions for terminology used in the descriptions of the plug-in APIs.

Plug-in
A dynamically loadable library that DB2 will load to access user-written authentication functions.

Implicit authentication
A connection to a database without specifying a user ID or a password.

Explicit authentication
A connection to a database in which both the user ID and password are specified.

Authid
An internal ID representing an individual or group to which authorities and privileges within the database are granted. Internally, DB2 authid is upper-cased and has a minimum of 8 characters (blank padded to 8 characters). Currently, DB2 requires authids, user IDs, passwords, group names, namespaces, and domain names that can be represented in 7-bit ASCII. The maximum length of an authid is 30 characters.

Local authorization
Authorization that is local to the server or client that implements it, that checks if a user is authorized to perform an action, other than connecting to the database that requires authorization, such as starting and stopping the database manager, turning DB2 trace on and off, or updating the database manager configuration.

Namespace
A collection or grouping of users within which individual user identifiers must be unique. Common examples include Windows® domains and Kerberos Realms. For example, within the Windows domain "usa.company.com" all user names must be unique. For example, "user1@usa.company.com". The same user ID in another domain, as in the case of "user1@canada.company.com", however refers to a different person. A fully qualified user identifier includes a user ID and namespace pair; for example, "user@domain.name" or "domain\user".

Related concepts:

- "Security plug-ins" on page 533

Group plug-in APIs

APIs for group retrieval plug-ins

For the group retrieval plug-in library, you will need to implement the following APIs:

```
SQL_API_RC SQL_API_FN db2secGroupPluginInit(
 db2int32 version,
 void *group_fns,
 db2secLogMessage *logMessage_fn,
 char **errormsg,
 db2int32  *errormsglen);
```

Note: The above function takes as input a pointer to a function, `*logMessage_fn`, with the following prototype:

```
SQL_API_RC (SQL_API_FN db2secLogMessage) (
  db2int32 level,
 void *data,
 db2int32 length);
```

```
SQL_API_RC SQL_API_FN db2secPluginTerm(char **errormsg,
      db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secGetGroupsForUser(
 const char *authid,
 db2int32  authidlen,
 const char *userid,
 db2int32  useridlen,
 const char *usernamespace,
 db2int32  usernamespacelen,
 db2int32  usernamespacetype,
 const char *dbname,
 db2int32  dbnamelen,
 const void *token,
 db2int32 tokentype,
 db2int32 location,
 const char *authpluginname,
 db2int32  authpluginname,
 char **grouplist,
 db2int32  *numgroups,
 char **errormsg,
 db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secDoesGroupExist(
  const char *groupname,
  db2int32  groupnamelen
  char **errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secFreeGroupListMemory(
  char *ptr,
  char **errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secFreeErrormsg(char *msgtobefree);
```

The only API that must be resolvable externally is `db2secGroupPluginInit()`. This function will take a `void *` parameter, which should be cast to the type:

```
typedef struct db2secGroupFunctions_1
{
  db2int32 version;
  db2int32 plugintype;
  SQL_API_RC (SQL_API_FN * db2secGetGroupsForUser) (
    const char *authid,
    db2int32  authidlen,
    const char *userid,
    db2int32  useridlen,
    const char *usernamespace,
    db2int32  usernamespacelen,
    db2int32  usernamespacetype,
    const char *dbname,
    db2int32  dbnamelen,
    const void *token,
    db2int32 tokentype,
    db2int32 location,
    const char *authpluginname,
    db2int32  authpluginnamelen,
    void **grouplist,
    db2int32  *numgroups,
    char **errormsg,
    db2int32  *errormsglen);

 SQL_API_RC (SQL_API_FN * db2secDoesGroupExist)(
    const char *groupname,
    db2int32  groupnamelen,
    char **errormsg,
    db2int32  *errormsglen);

  SQL_API_RC (SQL_API_FN * db2secFreeGroupListMemory)(
```

```
    void *ptr,
    char **errormsg,
    db2int32  *errormsglen);

  SQL_API_RC (SQL_API_FN * db2secFreeErrormsg)(
    char *msgtobefree);

 SQL_API_RC (SQL_API_FN * db2secPluginTerm)(
    char **errormsg,
    db2int32  *errormsglen);

} db2secGroupFunctions_1;
```

db2secGroupPluginInit() will assign the addresses for the rest of the externally available functions.

Note: The _1 indicates that this is the structure corresponding to version 1 of the API. Subsequent interface versions will have the extension _2, _3, and so on.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related tasks:

- "Deploying a group retrieval plug-in" on page 543

Related reference:

- "db2secGroupPluginInit - Initialize group plug-in" on page 562
- "db2secPluginTerm - Clean up group plug-in resources" on page 563
- "db2secGetGroupsForUser - Get list of groups for user" on page 564
- "db2secDoesGroupExist - Check if group exists" on page 567
- "db2secFreeGroupListMemory - Free group list memory" on page 568
- "db2secFreeErrormsg - Free error message memory" on page 569

db2secGroupPluginInit - Initialize group plug-in

This is the initialization function for the library that DB2 will call immediately after loading the plug-in library. The functions pointer should be cast to the appropriate group_functions structure for the interface version.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secGroupPluginInit(
  db2int32 version,
  void *group_fns,
  db2secLogMessage *logMessage_fn,
  char **errormsg,
  db2int32  *errormsglen);
```

Input:

db2int32 version
The highest version number of the API that DB2 will currently support.

*db2secLogMessage *logMessage_fn*
A pointer to a function provided by DB2. The plug-in can call this function to log additional error string to db2diag.log for debugging or informational purposes. The first parameter should use the define in

db2secPlugin.h and the last two parameters are the message string and its length. The defines to be used in the first parameter are:

```
#define DB2SEC_LOG_NONE       0   -  No logging
#define DB2SEC_LOG_CRITICAL   1   -  Severe Error encountered
#define DB2SEC_LOG_ERROR      2   -  Error encountered
#define DB2SEC_LOG_WARNING    3   -  Warning
#define DB2SEC_LOG_INFO       4   -  Informational
```

If you use the DB2SEC_LOG_INFO define, the message text will only show up in the db2diag.log if the *diaglevel* database manager configuration parameter is set to 4.

Output:

*void *group_fns*
A pointer to memory provided by DB2 for a db2secGroupFunction_1 structure. In future versions of DB2, there may be different versions of the APIs, so this should be cast to a pointer to the db2secGroupFunction_1 structure corresponding to the version of the API that the plug-in has implemented.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for group retrieval plug-ins" on page 560

db2secPluginTerm - Clean up group plug-in resources

This function will be called by DB2 just before it unloads the plug-in. It should do a proper cleanup of any resources the plug-in library holds, for instance, free any memory allocated by the plug-in, close files that are still open, and close network connections. The plug-in is responsible for keeping track of these resources in order to free them.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secPluginTerm(char **errormsg,
           db2int32  *errormsglen);
```

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "diaglevel - Diagnostic error capture level configuration parameter" in the *Administration Guide: Performance*
- "APIs for group retrieval plug-ins" on page 560

db2secGetGroupsForUser - Get list of groups for user

This function will be called by DB2 to get the list of groups to which a user belongs.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secGetGroupsForUser(
  const char *authid,
  db2int32  authidlen,
  const char *userid,
  db2int32  useridlen,
  const char *usernamespace,
  db2int32  usernamespacelen,
  db2int32  usernamespacetype,
  const char *dbname,
  db2inst32 dbnamelen,
  const void *token,
  db2int32 tokentype,
  db2int32 location,
  const char *authpluginname,
  db2int32  authpluginnamelen,
  void **grouplist,
  db2int32  *numgroups,
  char **errormsg,
  db2int32  *errormsglen);
```

Input:

*const char *authid*

The only input field that is provided by DB2. This field value is an SQL authid, therefore it is formatted to be a fully upper-cased character string with no trailing blanks. The plug-in must be able to return a list of groups to which the authid belongs without depending on the other input parameters. It is permissible to return a shortened or empty list if this cannot be determined.

If a user does not exist, the function should return `DB2SEC_PLUGIN_BADUSER`. DB2 does not treat the case of a user not existing as an error, since it is permissible for an authid to not have any groups associated with it. For example, when the `db2secGetAuthids` function returns an authid that does not exist on the operating system. The authid is not associated with any groups, however it can still be directly assigned privileges.

If the plug-in cannot return a complete list of groups from only the authid, then there will be some restrictions on certain SQL functions related to group support. Please refer to the note in this topic, titled ″Problems that may occur if an incomplete group list is returned″ for a list of possible problem scenarios.

db2int32 authidlen
Length of the authid.

*const char *userid*
This is the user ID corresponding to the authid. When this API is called on the server in a non-connect scenario, this will not be filled.

db2int32 useridlen
Length of the user ID.

*const void *token*
A pointer to data provided by the authentication plug-in. It is not seen by DB2. It provides the ability to the plug-in writer for coordinating user and group information, if desired. This may not be given in all cases, in which case it will be NULL. If the authentication plug-in used is GSS-API based, the token will be set to the GSS-API context handle (gss_ctx_id_t).

db2int32 tokentype
Indicates the type of data provided by the authentication plug-in. If the authentication plug-in used is GSS-API based, the token will be set to the GSS-API context handle (gss_ctx_id_t). If the authentication plug-in used is user ID/password based, it will be a generic type. See the following defines in `db2secPlugin.h`:

- #define DB2SEC_GENERIC 0 -- This indicates that the token is from a user ID/password based plug-in.
- #define DB2SEC_GSSAPI_CTX_HANDLE 1 -- This indicates that the token is from a GSS-API (including Kerberos) based plug-in.

db2int32 location
Indicates whether DB2 is to call the plug-in on the client side or server side. See the following define in `db2secPlugin.h`:

- #define DB2SEC_SERVER_SIDE 0 -- the group plug-in is being called on the database server.
- #define DB2SEC_CLIENT_SIDE 1 -- the group plug-in is being called on a client.

*const char *usernamespace*
The namespace from which the user ID was obtained. When the user ID is not available, this will not be filled.

db2int32 usernamespacelen
Length of the namespace field.

db2int32 usernamespacetype
The type of namespace. Possible values are: DB2SEC_NAMESPACE_SAM_COMPATIBLE (corresponding to a username style like torolab\myname″), or DB2SEC_NAMESPACE_USER_PRINCIPAL (corresponding to a username style like myname@torolab.ibm.com). Currently DB2 only supports DB2SEC_NAMESPACE_SAM_COMPATIBLE. When the user ID is not available, this will be filled with DB2SEC_USER_NAMESPACE_UNDEFINED. All the defines are located in `db2secPlugin.h`.

*const char *dbname*
This is the name of the database being connected to.

db2int32 dbnamelen
Length of the database name specified by *dbname*.

*const char *authpluginname*
This is the name of the authentication plug-in that provided the data in the token. The plug-in may use this information in determining the correct group memberships. This may not be given if the authid is not authenticated (for instance, if the authid does not match the current connected user).

db2int32 authpluginnamelen
Length of the authpluginname.

Output:

*void **grouplist*
The list of groups must be returned as a pointer to a section of memory allocated by the plug-in containing concatenated `varchars` (a `varchar` is a character array in which the first byte indicates the number of bytes following it). The length is an unsigned char and that limits the maximum length of a groupname to 255 characters. In other words, since we're using an unsigned char (1 byte) to indicate the length of the group name, the maximum length is 255. For example:

```
"\006GROUP1\007MYGROUP\008MYGROUP3"
```

Each group name should be a valid DB2 authid. The memory for this array must be allocated by the plug-in. The plug-in must therefore provide a function, such as the `db2secFreeGroupListMemory()` plug-in function that DB2 will call to free the memory.

*db2int32 *numgroups*
The number of groups contained in the grouplist.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Problems that may occur if an incomplete group list is returned from the API:

The following problems can occur if an incomplete group list is returned from this API to DB2 UDB:

- Embedded SQL application with DYNAMICRULES BIND (or DEFINEDBIND or INVOKEDBIND if the packages are running as a standalone application). DB2 checks for SYSADM membership and the application will fail if it is dependent on the implicit DBADM authority granted by being a member of the SYSADM group.
- Alternate authorization is provided in CREATE SCHEMA statement. Group lookup will be performed against the AUTHORIZATION NAME parameter if there are nested CREATE statements in the CREATE SCHEMA statement.
- Embedded SQL applications with DYNAMICRULES DEFINERUN/DEFINEBIND and the packages are running in a routine context. DB2 checks for SYSADM membership of the routine definer and the application will fail if it is dependent on the implicit DBADM authority granted by being a member of the SYSADM group.
- Processing a jar file in an MPP environment. In an MPP environment, the jar processing request is sent from the coordinator node with the session authid.

The catalog node received the requests and process the jar files based on the privilege of the session authid (the user executing the jar processing requests).
  - Install jar file. The session authid needs to have one of the following rights: SYSADM, DBADM, or CREATEIN (implicit or explicit on the jar schema). The operation will fail if the above rights are granted to group containing the session authid, but not explicitly to the session authid or if only SYSADM is held, since SYSADM membership is determined by membership in the group defined by a database configuration parameter.
  - Remove jar file. The session authid needs to have one of the following rights rights: SYSADM, DBADM, or DROPIN (implicit or explicit on the jar schema), or is the definer of the jar file. The operation will fail if the above rights are granted to group containing the session authid, but not explicitly to the session authid, and if the session authid is not the definer of the jar file or if only SYSADM is held since SYSADM membership is determined by membership in the group defined by a database configuration parameter.
  - Replace jar file. This is same as removing the jar file, followed by installing the jar file. Both of the above apply.
- Regenerate views. This is triggered by the ALTER TABLE, ALTER COLUMN, SET DATA TYPE VARCHAR/VARGRAPHIC statements, or during migration. DB2 checks for SYSADM membership of the view definer. The application will fail if it is dependent on the implicit DBADM authority granted by being a member of the SYSADM group.
- When SET SESSION_USER statement is issued. Subsequent DB2 operations are run under the context of the authid specified by this statement. These operations will fail if any required group privileges are owned by one of the SESSION_USER's groups, but are not explicitly granted to the SESSION_USER authid.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for group retrieval plug-ins" on page 560

db2secDoesGroupExist - Check if group exists

This function will be used to determine if an authid represents a group. The function should return DB2SEC_PLUGIN_OK, to indicate success, if the groupname exists. It should return DB2SEC_PLUGIN_INVALIDUSERORGROUP if the group name is not valid. It is also permissible for the API to return DB2SEC_PLUGIN_GROUPSTATUSNOTKNOWN if it is impossible to determine if the input is a valid group. If invalid group or group not known is returned, DB2 might not be able to determine whether the authid is a group or user when issuing the GRANT statement without the keywords USER and GROUP, which would result in the error SQLCODE -569, SQLSTATE 56092 being returned to the user.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secDoesGroupExist(
 const char *groupname,
 db2int32  groupnamelen,
 char **errormsg,
 db2int32  *errormsglen
);
```

Input:

*const char *groupname*
An authid, upper-cased, with no trailing blanks.

db2int32 groupnamelen
Length of the groupname.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for group retrieval plug-ins" on page 560

db2secFreeGroupListMemory - Free group list memory

This function tells the plug-in library that the memory pointed to by `ptr` is no longer needed by DB2. The plug-in needs to free this memory.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secFreeGroupListMemory(
 void *ptr
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*void *ptr*
Pointer to the memory to be freed.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for group retrieval plug-ins" on page 560

db2secFreeErrormsg - Free error message memory

This function will be called by DB2 to free the memory used to hold an error message from a previous call to a plug-in API. This is the only API that does not also return an error message. If this function returns an error, DB2 will log it and continue.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secFreeErrormsg(char *msgtobefree);
```

Input:

*char *msgtobefree*
A pointer to the error message allocated from a previous API call.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for group retrieval plug-ins" on page 560

User authentication plug-in APIs

APIs for user ID/password authentication plug-in

For the user ID/password plug-in library, you will need to implement the following client-side APIs:

```
SQL_API_RC SQL_API_FN db2secClientAuthPluginInit(
  db2int32 version,
  void *client_fns,
  db2secLogMessage *logMessage_fn,
  char **errormsg,
  db2int32  *errormsglen);
```

Note: The above function takes as input a pointer to a function, `*logMessage_fn`, with the following prototype:

```
SQL_API_RC (SQL_API_FN db2secLogMessage) (
  db2int32 level,
  void *data,
  db2int32 length);
```

```
SQL_API_RC SQL_API_FN db2secClientAuthPluginTerm(
  char **errormsg,
  db2int32  *errormsglen);

/* Only used for gssapi: */
db2secGenerateInitialCred(
  const char *userid,
  db2int32  useridlen,
  const char *usernamespace,
  db2int32  usernamespacelen,
  db2int32  usernamespacetype,
  const char *password,
  db2int32  passwordlen,
  const char *newpassword,
```

```
|                           db2int32  newpasswordlen,
|                           const char *dbname,
|                           db2int32  dbnamelen,
|                           gss_cred_id_t *pGSSCredHandle,
|                           void **initInfo,
|                           char **errormsg,
|                           db2int32  *errormsglen);
|
|                         /* Optional */
|                         SQL_API_RC SQL_API_FN db2secRemapUserid(
|                           char userid[DB2SEC_MAX_USERID_LENGTH],
|                           db2int32  *useridlen,
|                           db2int32  useridtype,
|                           char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
|                           db2int32  *usernamespacelen,
|                           db2int32  *usernamespacetype,
|                           char password[DB2SEC_MAX_PASSWORD_LENGTH],
|                           db2int32  *passwordlen,
|                           char newpassword[DB2SEC_MAX_PASSWORD_LENGTH],
|                           db2int32  *newpasswordlen,
|                           const char *dbname,
|                           db2int32  dbnamelen,
|                           char **errormsg,
|                           db2int32  *errormsglen);
|                         SQL_API_RC SQL_API_FN db2secGetDefaultLoginContext(
|                           char authid[DB2SEC_MAX_AUTHID_LENGTH],
|                           db2int32  *authidlen,
|                           char userid[DB2SEC_MAX_USERID_LENGTH],
|                           db2int32  *useridlen,
|                           db2int32 useridtype,
|                           char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
|                           db2int32  *usernamespacelen,
|                           db2int32  *usernamespacetype,
|                           const char *dbname,
|                           db2int32  dbnamelen,
|                           void **token,
|                           char **errormsg,
|                           db2int32  *errormsglen);
|                         SQL_API_RC SQL_API_FN db2secValidatePassword(
|                           const char *userid,
|                           db2int32  useridlen,
|                           const char *usernamespace,
|                           db2int32  usernamespacelen,
|                           db2int32  usernamespacetype,
|                           const char *password,
|                           db2int32  passwordlen,
|                           const char *newpassword,
|                           db2int32  newpasswordlen,
|                           const char *dbname,
|                           db2int32  dbnamelen,
|                           db2Uint32   connection_details,
|                           void **token,
|                           char **errormsg,
|                           db2int32  *errormsglen);
|                         /* This is only for GSS-API */
|                         SQL_API_RC SQL_API_FN db2secProcessServerPrincipalName(
|                           const void *name,
|                           db2int32 nameLen,
|                           gss_name_t *gssName,
|                           char **errormsg,
|                           db2int32  *errormsglen);
```

```
/* Functions to free memory held by the DLL */
SQL_API_RC SQL_API_FN db2secFreeToken(
  void *token,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC SQL_API_FN db2secFreeErrormsg(char *errormsg);

SQL_API_RC SQL_API_FN db2secFreeInitInfo(
  void *initInfo,
  char **errormsg,
  db2int32  *errormsglen);
```

The only API that must be resolvable externally is `db2secClientAuthPluginInit()`. This function will take a `void *` parameter, which should be cast to either:

```
typedef struct db2secUseridPasswordClientAuthFunctions_1
{
   db2int32 version;
   db2int32 plugintype;

SQL_API_RC (SQL_API_FN * db2secGetDefaultLoginContext)(
  char authid[DB2SEC_MAX_AUTHID_LENGTH],
  db2int32  *authidlen,
  char userid[DB2SEC_MAX_USERID_LENGTH],
  db2int32  *useridlen,
  db2int32 useridtype,
  char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
  db2int32  *usernamespacelen,
  db2int32  *usernamespacetype,
  const char *dbname,
  db2int32  dbnamelen,
  void **token,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secRemapUserid)(  // Optional
  char userid[DB2SEC_MAX_USERID_LENGTH],
  db2int32  *useridlen,
  char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
  db2int32  *usernamespacelen,
  db2int32  *usernamespacetype,
  char password[DB2SEC_MAX_PASSWORD_LENGTH],
  db2int32  *passwordlen,
  char newpassword[DB2SEC_MAX_PASSWORD_LENGTH],
  db2int32  *newpasswordlen,
  const char *dbname,
  db2int32  dbnamelen,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secValidatePassword)(
  const char *userid,
  db2int32  useridlen,
  const char *usernamespace,
  db2int32  usernamespacelen,
  db2int32  usernamespacetype,
  const char *password,
  db2int32  passwordlen,
  const char *newpassword,
  db2int32  newpasswordlen,
  const char *dbname,
  db2int32  dbnamelen,
  db2Uint32   connection_details,
  void **token,
  char **errormsg,
  db2int32  *errormsglen);
```

```
SQL_API_RC (SQL_API_FN * db2secFreeToken)(
  void **token,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secFreeErrormsg)(char *errormsg);


SQL_API_RC (SQL_API_FN * db2secClientAuthPluginTerm)(
  char **errormsg,
  db2int32  *errormsglen);
}
```

or

```
typedef struct db2secGssapiClientAuthFunctions_1
{
  db2int32 version;
  db2int32 plugintype;

SQL_API_RC (SQL_API_FN * db2secGetDefaultLoginContext) (
  char authid[DB2SEC_MAX_AUTHID_LENGTH],
  db2int32  *authidlen,
  char userid[DB2SEC_MAX_USERID_LENGTH],
  db2int32  *useridlen,
  db2int32 useridtype,
  char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
  db2int32  *usernamespacelen,
  db2int32  *usernamespacetype,
  const char *dbname,
  db2int32  dbnamelen,
  void **token,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secProcessServerPrincipalName) (
  const void *data,
  gss_name_t *gssName,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secGenerateInitialCred) (
  const char *userid,
  db2int32 useridlen,
  const char *usernamespace,
  db2int32 usernamespacelen,
  db2int32 usernamespacetype,
  const char *password,
  db2int32 passwordlen,
  const char *newpassword,
  db2int32 newpasswordlen,
  const char *dbname,
  db2int32 dbnamelen,
  gss_cred_id_t *pGSSCredHandle,
  void **initInfo,
  char **errormsg,
  db2int32 *errormsglen);

SQL_API_RC (SQL_API_FN * db2secFreeToken)(
  void *token,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secFreeErrormsg)(char *errormsg);


SQL_API_RC (SQL_API_FN * db2secFreeInitInfo) (
```

```
  void *initInfo,
  char **errormsg,
  db2int32  *errormsglen);

SQL_API_RC (SQL_API_FN * db2secClientAuthPluginTerm) (
  char **errormsg,
  db2int32  *errormsglen);

/* GSS-API specific functions -- refer to db2secPlugin.h
   for parameter list*/

  OM_uint32 (SQL_API_FN * gss_init_sec_context )(<parameter list>);
  OM_uint32 (SQL_API_FN * gss_delete_sec_context )(<parameter list>);
  OM_uint32 (SQL_API_FN * gss_display_status )(<parameter list>);
  OM_uint32 (SQL_API_FN * gss_release_buffer )(<parameter list>);
  OM_uint32 (SQL_API_FN * gss_release_cred )(<parameter list>);
  OM_uint32 (SQL_API_FN * gss_release_name )(<parameter list>);
  }
```

You should use `db2secUseridPasswordClientAuthFunctions_1` if you are writing an user ID/password plug-in. If you are writing a GSS-API (including Kerberos) plug-in, you should use `db2secGssapiClientAuthFunctions_1`.

For the user ID/password plug-in library, you will need to implement the following server-side APIs:

```
db2secServerAuthPluginInit(
  db2int32 version,
  void *server_fns,
  db2secGetConDetails *getConDetails_fn,
  db2secLogMessage *logMessage_fn,
  char **errormsg,
  db2int32  *errormsglen);
```

The above function takes as input a pointer to a function, `*logMessage_fn`, and a function, `*getConDetails_fn`, with the following prototypes:

```
SQL_API_RC (SQL_API_FN db2secLogMessage) (
  db2int32 level,
  void *data,
  db2int32 length);

SQL_API_RC (SQL_API_FN db2secGetConDetails)(
  db2int32 conDetailsVersion,
  const void *pConDetails);
```

This function in turn, takes as its second parameter, `pConDetails`, a pointer to a structure defined as follows:

```
typedef struct db2sec_con_details_1
{
  db2int32 clientProtocol;
  db2Uint32 clientIPAddress;
  db2Uint32 connect_info_bitmap;
  db2int32 dbnameLen;
  char dbname[DB2SEC_MAX_DBNAME_LENGTH + 1];
} db2sec_con_details_1;
```

See the detailed description section for an explanation of this function and structure.

```
db2secServerAuthPluginTerm(
  char **errormsg,
  db2int32  *errormsglen);
```

```
SQL_API_RC SQL_API_FN db2secValidatePassword(
  const char *userid,
  db2int32  useridlen,
  const char *usernamespace,
  db2int32  usernamespacelen,
  db2int32  usernamespacetype,
  const char *password,
  db2int32  passwordlen,
  const char *newpasswd,
  db2int32  newpasswdlen,
  const char *dbname,
  db2int32  dbnamelen,
  db2Uint32  connection_details,
  void **token,
  char **errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secGetAuthIDs(
  const char *userid,
  db2int32  useridlen,
  const char *usernamespace,
  db2int32  usernamespacelen,
  db2int32  usernamespacetype,
  const char *dbname,
  db2int32  dbnamelen,
  void **token,
  char SystemAuthid[DB2SEC_MAX_AUTHID_LENGTH],
  db2int32  SystemAuthidlen,
  char InitialSessionAuthID[DB2SEC_MAX_AUTHID_LENGTH],
  db2int32  *InitialSessionAuthIdlen,
  char username[DB2SEC_MAX_USERID_LENGTH],
  db2int32  *usernamelen,
  db2int32 *initsessionidtype,
  char **errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secDoesAuthIDExist(
  const char *authid,
  db2int32 authidlen,
  const char *errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secFreeToken(
  void *token,
  char **errormsg,
  db2int32  *errormsglen);
SQL_API_RC SQL_API_FN db2secFreeErrormsg(char *errormsg);
```

The only API that must be resolvable externally is `db2secServerAuthPluginInit()`. This function will take a `void *` parameter, which should be cast to either:

```
typedef struct db2secUseridPasswordServerAuthFunctions_1
{
  db2int32 version;
  db2int32 plugintype;

  /* parameter lists left blank for readability
    see above for parameters */
 SQL_API_RC (SQL_API_FN * db2secValidatePassword)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secGetAuthIDs)(<parameter list);
  SQL_API_RC (SQL_API_FN * db2secDoesAuthIDExist)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secFreeToken)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secFreeErrormsg)(<parameter list>);
 SQL_API_RC (SQL_API_FN * db2secServerAuthPluginTerm)();
} userid_password_server_auth_functions;
```

or

```
typedef struct db2secGssapiServerAuthFunctions_1
{
 db2int32 version;
  db2int32 plugintype;
  gss_buffer_desc serverPrincipalName;
  gss_cred_id_t ServerCredHandle;
  SQL_API_RC (SQL_API_FN * db2secGetAuthIDs)(<parameter list);
  SQL_API_RC (SQL_API_FN * db2secDoesAuthIDExist)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secFreeToken)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secFreeErrormsg)(<parameter list>);
  SQL_API_RC (SQL_API_FN * db2secServerAuthPluginTerm)();

 /* GSS-API specific functions
 refer to db2secPlugin.h for parameter list*/
 OM_uint32 (SQL_API_FN * gss_accept_sec_context )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_display_name )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_delete_sec_context )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_display_status )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_release_buffer )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_release_cred )(<parameter list>);
 OM_uint32 (SQL_API_FN * gss_release_name )(<parameter list>);

} gssapi_server_auth_functions;
```

You should use `db2secUseridPasswordServerAuthFunctions_1` if you are writing an user ID/password plug-in. If you are writing a GSS-API (including Kerberos) plug-in, you should use `db2secGssapiServerAuthFunctions_1`.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related tasks:

- "Deploying a user ID/password plug-in" on page 543

Related reference:

- "db2secGetGroupsForUser - Get list of groups for user" on page 564
- "db2secClientAuthPluginInit - Initialize client authentication plug-in" on page 576
- "db2secClientAuthPluginTerm - Clean up client authentication plug-in resources" on page 577
- "db2secRemapUserid - Remap user ID and password" on page 577
- "db2secGetDefaultLoginContext - Get default login context" on page 579
- "db2secGenerateInitialCred - Generate initial credentials" on page 580
- "db2secValidatePassword - Validate password" on page 582
- "db2secProcessServerPrincipalName - Process service principal name returned from server" on page 584
- "db2secFreeToken - Free memory held by token" on page 585
- "db2secFreeInitInfo - Clean up resources held by db2secGenerateInitialCred()" on page 586
- "db2secServerAuthPluginInit - Initialize server authentication plug-in" on page 587
- "db2secServerAuthPluginTerm - Clean up server authentication plug-in resources" on page 588
- "db2secGetAuthIDs - Get authentication IDs" on page 589

- "db2secDoesAuthIDExist - Check if authentication ID exists" on page 591

db2secClientAuthPluginInit - Initialize client authentication plug-in

This is the initialization function for the plug-in library that DB2 will call immediately after loading the library. The `functions` pointer should be cast to the appropriate `client_auth_functions` structure for the interface version.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secClientAuthPluginInit(
 db2int32 version,
 void *client_fns,
 db2secLogMessage *logMessage_fn,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

db2int32 version
The highest version number of the API that DB2 will currently support.

*db2secLogMessage *logMessage_fn*
A pointer to a function provided by DB2. The plug-in can call this function to log additional error string to `db2diag.log` for debugging or informational purposes. The first parameter should use the define in `db2secPlugin.h` and the last two parameters are the message string and its length. The defines to be used in the first parameter are:

```
#define DB2SEC_LOG_NONE       0   -  No logging
#define DB2SEC_LOG_CRITICAL   1   -  Severe Error encountered
#define DB2SEC_LOG_ERROR      2   -  Error encountered
#define DB2SEC_LOG_WARNING    3   -  Warning
#define DB2SEC_LOG_INFO       4   -  Informational
```

If you use the DB2SEC_LOG_INFO define, the message text will only show up in the `db2diag.log` if the *diaglevel* database manager configuration parameter is set to 4.

Output:

*void *client_fns*
A pointer to memory provided by DB2 for a `client_auth_functions` structure. In future versions of DB2, there can be different versions of the APIs, so this should be cast to a pointer to the `client_auth_functions` structure corresponding to the version of the API that the plug-in implements.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "diaglevel - Diagnostic error capture level configuration parameter" in the *Administration Guide: Performance*
- "APIs for user ID/password authentication plug-in" on page 569

db2secClientAuthPluginTerm - Clean up client authentication plug-in resources

This function will be called by DB2 just before it unloads the plug-in. It should do a proper cleanup of any resources the plug-in library holds, for instance, free any memory allocated by the plug-in, close files that are still open, and close network connections. The plug-in is responsible for keeping track of these resources in order to free them.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secClientAuthPluginTerm(
 char **errormsg
 db2int32  *errormsglen);
```

Output:

*char **errormsg*
: A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
: A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secRemapUserid - Remap user ID and password

This function will be called on the client side to provide the ability to remap a given user ID and password (and possibly new password and usernamespace) to different values from those given at connect time. DB2 will only call this function if at least a user ID and a password are supplied at connect time. This prevents a plug-in from remapping a user ID by itself to a user ID/password pair. This function is optional and will not be called if it is not provided.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secRemapUserid(
  char userid[DB2SEC_MAX_USERID_LENGTH],
  db2int32  *useridlen,
  char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
  db2int32  *usernamespacelen,
  db2int32  *usernamespacetype,
  char password[DB2SEC_MAX_PASSWORD_LENGTH],
  db2int32  *passwordlen,
  char newpassword[DB2SEC_MAX_PASSWORD_LENGTH],
  db2int32  *newpasswordlen,
```

```
  const char *dbname,
  db2int32  dbnamelen,
  char **errormsg,
  db2int32  *errormsglen);
```

Input:

*const char *dbname*
The name of the database to which the client is connecting.

db2int32 dbnamelen
Length of the dbname.

Input/output:

char userid[DB2SEC_MAX_USERID_LENGTH]
The user ID to be remapped. If there is an input user ID value, then there must be an output user ID value that can be the same or different from the input user ID value. If there is no input user ID value, then the plug-in should not return an output user ID value.

*db2int32 *useridlen*
Length of the user ID returned in the userid parameter.

char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH]
The namespace the user ID came from. It is optional to remap this. If usernamespace was not provided as input to this function, and the function does provide a value as output, then the usernamespace will only be used by DB2 for CLIENT authentication and be disregarded for other authentication types.

*db2int32 *usernamespacelen*
Old and new length of the usernamespace. Under the limitation that the usernamespacetype must be DB2SEC_NAMESPACE_SAM_COMPATIBLE, the maximum length supported in the current version of DB2 is 15 bytes.

*db2int32 *usernamespacetype*
Old and new namespacetype. In the current version of DB2, the only supported namespace type is DB2SEC_NAMESPACE_SAM_COMPATIBLE.

char password[DB2SEC_MAX_PASSWORD_LENGTH]
The password to be remapped. If a password was passed as input, then the plug-in must output the new password, which can be a different than the original password. If there is no password passed in as input, then the plug-in should not return an output password.

*db2int32 *passwordlen*
Length of the password.

char newpassword[DB2SEC_MAX_PASSWORD_LENGTH]
A new password if the password is to be changed.

Note: This is the new password that will be passed by DB2 into the newpassword field of the db2secValidatePassword function on the client or the server (depending on the value of the AUTHENTICATION dbm configuration parameter). If a new password was passed as input, then there must an output new password and can be a different new password. If there is no new password passed in as input, then the plug-in should not return an output new password.

*db2int32 *newpasswordlen*
Length of the new password.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secGetDefaultLoginContext - Get default login context

This function is called by DB2 to determine the user associated with the default login context, in other words, to determine the DB2 authid of the user invoking a DB2 command without explicitly specifying a user ID (either an implicit authentication to a database, or a local authorization). This function must return both an authid and a user ID.

C API syntax:

```
db2secGetDefaultLoginContext(
 char authid[DB2SEC_MAX_AUTHID_LENGTH],
 db2int32  *authidlen,
 char userid[DB2SEC_MAX_USERID_LENGTH],
 db2int32  *useridlen,
 db2int32  useridtype,
 char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH],
 db2int32  *usernamespacelen,
 db2int32  *usernamespacetype,
 const char *dbname,
 db2int32  dbnamelen,
 void **token,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*const char *dbname*
This will contain the name of the database being connected to if this call is being used in the context of a database connection. For local authorization actions or instance attachments, this parameter will be NULL.

db2int32 dbnamelen
Length of the dbname.

db2int32 useridtype
Specifies if DB2 wants the real or effective user of the process.

Output:

char authid[DB2SEC_MAX_AUTHID_LENGTH]
The field in which the authid should be returned. The returned value must conform to DB2 authid naming questions, or the user will not be authorized to perform the requested action.

*db2int32 *authidlen*
Length of the authid returned.

char userid[DB2SEC_MAX_USERID_LENGTH]
The field in which the user ID should be returned.

*db2int32 *useridlen*
Length of the user ID returned.

*void **token*
This is a pointer to data allocated by the plug-in that it will want to pass to subsequent authentication calls in the plug-in, or possibly to the group retrieval plug-in. The structure of this data is to be decided by the plug-in writer.

char usernamespace[DB2SEC_MAX_USERNAMESPACE_LENGTH]
Length of the returned namespace. Under the limitation that the usernamespacetype must be DB2SEC_NAMESPACE_SAM_COMPATIBLE, the maximum length supported in the current version of DB2 is 15 bytes.

*db2int32 *usernamespacelen*
Length of the namespace returned. Under the limitation that the usernamespacetype must be DB2SEC_NAMESPACE_SAM_COMPATIBLE, the maximum length supported in the current version of DB2 is 15 bytes.

*db2int32 *usernamespacetype*
As specified above. In the current version of DB2, the only supported namespace type is DB2SEC_NAMESPACE_SAM_COMPATIBLE.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "authentication - Authentication type configuration parameter" in the *Administration Guide: Performance*
- "APIs for user ID/password authentication plug-in" on page 569

db2secGenerateInitialCred - Generate initial credentials

This function will obtain the initial GSS-API credentials based on the user ID and password that are passed in. For Kerberos this will be the TGT. The credential handle that is returned in `pGSSCredHandle` is the handle that will be used with `gss_init_sec_context()` and must be either an INITIATE or BOTH credential. This function will only be called when a user ID, and possibly a password are supplied.

Otherwise, DB2 will specify GSS_C_NO_CREDENTIAL when calling gss_init_sec_context() to signify that the default credential obtained from the current login context is to be used.

C API syntax:

```
db2secGenerateInitialCred(
 const char *userid,
 db2int32  useridlen,
 const char *usernamespace,
 db2int32  usernamspacelen,
 db2int32  usernamespacetype,
 const char *password,
 db2int32  passwordlen,
 const char *newpassword,
 db2int32  newpasswordlen,
 const char *dbname,
 db2int32  dbnamelen,
 gss_cred_id_t *pGSSCredHandle,
 void **initInfo,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*const char *userid*
The user ID whose password is to be verified.

db2int32 useridlen
Length of the user ID.

*const char *usernamespace*
The namespace from which the user ID was obtained.

db2int32 usernamespacelen
Length of the namespace field.

db2int32 usernamespacetype
The type of namespace.

*const char *password*
The password to be verified. This will be unencrypted by DB2 before being passed to the plug-in.

db2int32 passwordlen
Length of the newpassword.

*const char *newpassword*
A new password if the password is to be changed. If no change is requested, this will be NULL. If this is non-NULL, the function should validate the old password before changing to the new password. The plug-in does not have to honour a request to change the password, but if it does not, it should immediately return DB2SEC_PLUGIN_CHANGEPASSWORD_NOTSUPPORTED without validating the old password.

db2int32 newpasswordlen
Length of the newpassword.

*const char *dbname*
The name of the database being connected to. This function is free to ignore this, or this function can return DB2SEC_PLUGIN_CONNECTION_DISALLOWED if it wishes to restrict access to certain databases to users who otherwise have valid passwords.

db2int32 dbnamelen
Length of the dbname.

Output:

*gss_cred_id_t *pGSSCredHandle*
Pointer to the GSS-API credential handle.

*void **initInfo*
A pointer to data that is opaque to DB2. The plug-in can use this to maintain a list of resources that are allocated in the process of generating the credential handle. DB2 will call `db2secFreeInitInfo()` at the end of authentication, at which point the plug-in can then free these resources. If the plug-in does not need to maintain such a list, then it should return NULL.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

Note: For this API, error messages should not be created if the return value indicates a bad user ID or password. An error message should only be returned if there is an internal error in the API that prevented it from returning properly.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secValidatePassword - Validate password

This function will provide a user ID-and-password-style authentication method during a database connect operation.

Note: The plug-in code will be run with the privileges of the client application. The plug-in writer should take this into consideration if authentication requires special privileges (such as root).

This API should return DB2SEC_PLUGIN_OK (success) if the password is valid, or an error code such as DB2SEC_PLUGIN_BADPWD if the password is invalid.

This API will only be called on the client side if *authentication* is set to CLIENT.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secValidatePassword(
 const char *userid,
 db2int32  useridlen,
 const char *usernamespace,
 db2int32  usernamespacelen,
 db2int32  usernamespacetype,
 const char *password,
 db2int32  passwordlen,
```

```
 const char *newpassword,
 db2int32  newpasswordlen,
 const char *dbname,
 db2int32  dbnamelen,
  db2Uint32   connection_details,
  void **token,
  char **errormsg,
  db2int32  *errormsglen);
```

Input:

*const char *userid*
: The user ID whose password is to be verified.

db2int32 useridlen
: Length of the user ID.

*const char *password*
: The password to be verified. This will be unencrypted by DB2 before being passed in.

db2int32 passwordlen
: Length of the password given.

*const char *newpassword*
: A new password, if the password is to be changed. If no change is requested, this parameter will be NULL. If this parameter is not NULL, the function should validate the old password before changing to the new password. The plug-in does not have to fulfill a request to change the password, but if it does not, it should immediately return DB2SEC_PLUGIN_CHANGEPASSWORD_NOTSUPPORTED without validating the old password.

db2int32 newpasswordlen
: Length of the newpassword.

*const char *dbname*
: The name of the database being connected to. The function is free to ignore this, or it can return DB2SEC_PLUGIN_CONNECTIONREFUSED if it has a policy of restricting access to certain databases to users who otherwise have valid passwords.

db2int32 dbnamelen
: Length of the dbname.

db2int32 usernamespace
: The namespace from which the user ID was obtained.

db2int32 usernamespacelen
: Length of the namespace field.

db2int32 usernamespacetype
: The type of namespace. Possible values are: DB2SEC_NAMESPACE_SAM_COMPATIBLE (corresponding to a username style like torolab\myname"), or DB2SEC_NAMESPACE_USER_PRINCIPAL (corresponding to a username style like myname@torolab.ibm.com). Currently DB2 only supports DB2SEC_NAMESPACE_SAM_COMPATIBLE. When the user ID is not available, this will be filled with DB2SEC_USER_NAMESPACE_UNDEFINED. All the defines are located in `db2secPlugin.h`.

db2Uint32 connection_details
: A bit field currently consisting of 2 fields:

- 1 bit will indicate whether the connection is local (using ipc or connect from one of the nodes in the db2nodes.cfg in the EEE cluster), or remote (through network or loopback). This will give the plug-in the ability to decide whether clients on the same machine can connect to the DB2 server without a password. Currently, DB2 always allows local connections without a password from clients on the same machine (assuming the client has connect privileges).
- 1 bit will indicate whether the source of the user ID is the default from db2secGetDefaultLoginContext, or was explicitly provided during the connect. The bit values are defined in db2secPlugin.h:

  ```
  #define DB2SEC_USERID_FROM_OS 0x00000001
  ```

  DB2SEC_USERID_FROM_OS indicates user ID is obtained from OS and not explicitly given on the connect statement.

  ```
  #define DB2SEC_CONNECTION_ISLOCAL 0x00000002
  ```

  DB2SEC_CONNECTION_ISLOCAL indicates a local connection.

  ```
  #define DB2SEC_VALIDATING_ON_SERVER_SIDE 0x0000004
  ```

  DB2SEC_VALIDATING_ON_SERVER_SIDE indicates whether DB2 is calling from the server side for validating password.

  DB2's default behavior for an implicit authentication is to allow the connection without any password validation. However, plug-in developers have the option of disallowing implicit authentication by returning a DB2SEC_PLUGIN_BADPASSWORD error.

*void **token*
: A pointer to data which can be passed into subsequent plug-in API calls (db2secGetAuthIDs, db2secGetGroupsForUser) during this connection.

Output:

*char **errormsg*
: A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
: A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secProcessServerPrincipalName - Process service principal name returned from server

This function will process the service principal name returned from the server and return the principal name in the gss_name_t internal format to be used with gss_init_sec_context(). This function will also be called to process the service principal name cataloged with the db directory in the case of Kerberos authentication. Normally, this conversion employs the use of the

gss_import_name() API. Once the context has been established, the gss_name_t object will be freed through the call to gss_release_name(). The function will return DB2SEC_PLUGIN_OK if gssName points to a valid GSS name; a DB2SEC_PLUGIN_BAD_PRINCIPAL_NAME error code will be returned if the principal name is invalid.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secProcessServerPrincipalName(
 const void *name,
 db2int32 nameLen,
 gss_name_t *gssName,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*const void *name*
Text name of the service principal in GSS_C_NT_USER_NAME format, e.g., service/host@REALM.

db2int32 nameLen
Length of the text service principal name.

Output:

*gss_name_t *gssName*
Pointer to the output service principal name in the GSS-API internal format

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secFreeToken - Free memory held by token

This function will be called by DB2 when it no longer needs the memory held by token. The plug-in must free the memory.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secFreeToken(
 void *token
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*void *token*
Pointer to the memory to be freed.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secFreeInitInfo - Clean up resources held by db2secGenerateInitialCred()

This function will free any resource allocated by `db2secGenerateInitialCred()`. This can include for example, handles to underlying mechanism contexts or a credential cache created for the GSS-API credential cache.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secFreeInitInfo(
 void *initinfo,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*void *initinfo*
A pointer to data opaque to DB2. The pointer is used to maintain a list of resources that are allocated in the process of generating the credential handle. These resources will be freed by calling this API.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secServerAuthPluginInit - Initialize server authentication plug-in

This is the initialization function for the library that DB2 will call immediately after loading the library. The `functions` pointer should be cast to the appropriate `server_auth_functions` structure for the interface version. In the case of GSS-API, the plug-in is responsible for filling in the server's principal name in the `serverPrincipalName` variable inside the `gssapi_server_auth_functions` structure at initialization time and providing the server's credential handle in the *serverCredHandle* variable. The freeing of the memory allocated to hold the principal name and the credential handle is the responsibility of the `db2secServerAuthPluginTerm()` cleanup function.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secServerAuthPluginInit(
 db2int32 version,
  void *server_fns,
  db2secGetConDetails *getConDetails_fn,
  db2secLogMessage *logMessage_fn,
  char **errormsg,
  db2int32  *errormsglen);
```

Input:

db2int32 version
The highest version number of the API that DB2 will currently support.

*db2secGetConDetails *getConDetails_fn*
This is a pointer to a function provided by DB2. The plug-in can call this function in any one of the other authentication APIs to obtain details related to the database connection. These details will include information about the communication mechanism associated with the connection (such as the IP address, in the case of TCP/IP), which the plug-in writer might need to reference when making authentication decisions. For instance, the plug-in could disallow a connection for a particular user, unless that user is connecting from a particular IP address. The use of this callback is optional.

If the callback is called in a situation not involving a database connection, this function will return `DB2SEC_PLUGIN_NO_CON_DETAILS`, otherwise, this function will return 0 on success.

The parameter `getConDetails_fn` takes two input parameters, a pointer to the `db2sec_con_details` structure, and a version number indicating which `db2sec_con_details` structure to use. The current version number is 1. Upon a successful return, the `db2sec_con_details` structure will be filled out with the following details:

- The protocol used for the connection to the server. The listing of protocol definitions can be found in file `sqlenv.h` *lpar;SQL_PROTOCOL_*).
- The TCP/IP address of the inbound connect to the server if the protocol is TCP/IP.
- The database name the client is attempting to connect to. This will not be set for instance attachments.
- A connection information bit-map that contains the same details as documented in the `connection_details` parameter of the `db2secValidatePassword()` API.

*db2secLogMessage *logMessage_fn*
A pointer to a function provided by DB2. The plug-in can call this function to log additional error string to `db2diag.log` for debugging or informational purposes. The first parameter should use the define in `db2secPlugin.h` and the last two parameters are the message string and its length. The defines to be used in the first parameter are:

```
#define DB2SEC_LOG_NONE       0   -  No logging
#define DB2SEC_LOG_CRITICAL   1   -  Severe Error encountered
#define DB2SEC_LOG_ERROR      2   -  Error encountered
#define DB2SEC_LOG_WARNING    3   -  Warning
#define DB2SEC_LOG_INFO       4   -  Informational
```

If you use the DB2SEC_LOG_INFO define, the message text will only show up in the `db2diag.log` if the *diaglevel* database manager configuration parameter is set to 4.

Output:

*void *server_fns*
A pointer to memory provided by DB2 for a `server_auth_functions` structure. In future versions of DB2, there can be different versions of the APIs, so this should be cast to a pointer to the `server_auth_functions` structure corresponding to the version of the API that the plug-in implements.

Inside the `server_auth_functions`, the `plugintype` variable should be set to one of DB2SEC_PLUGIN_TYPE_USERID_PASSWORD, DB2SEC_PLUGIN_TYPE_GSSAPI, or DB2SEC_PLUGIN_TYPE_KERBEROS. Other values can be defined in future versions of the API.

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "diaglevel - Diagnostic error capture level configuration parameter" in the *Administration Guide: Performance*
- "APIs for user ID/password authentication plug-in" on page 569

db2secServerAuthPluginTerm - Clean up server authentication plug-in resources

This function will be called by DB2 just before it unloads the plug-in. It should do a proper cleanup of any resources the plug-in library holds, for instance, free any memory allocated by the plug-in, close files that are still open, and close network connections. The plug-in is responsible for keeping track of these resources in order to free them.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secServerAuthPluginTerm(char **errormsg,
 db2int32  *errormsglen);
```

Output:

*char **errormsg*
: A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
: A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secGetAuthIDs - Get authentication IDs

This function returns an SQL authid for an authenticated user. This will be called during database connections for both user ID/password and GSS-API authentication methods.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secGetAuthIDs(
 const char *userid,
 db2int32  useridlen,
 const char *usernamespace,
 db2int32  usernamespacelen,
 db2int32  usernamespacetype,
 const char *dbname,
 db2int32  dbnamelen,
 void **token,
 char SystemAuthID[DB2SEC_MAX_AUTHID_LENGTH],
 db2int32  *SystemAuthIDlen,
 char InitialSessionAuthID[DB2SEC_MAX_AUTHID_LENGTH],
 db2int32  *InitialSessionAuthIDlen,
 char username[DB2SEC_MAX_USERID_LENGTH],
 db2int32  *usernamelen,
 db2int32 *initsessionidtype,
 char **errormsg,
 db2int32  *errormsglen);
```

Input:

*const char * userid*
: The authenticated user. This will be blank for GSS-API.

db2int32 useridlen
: Length of the user ID.

*void **token*
: Data that the plug-in might pass to the `db2secGetGroupsForUser` call. For GSS-API, this is a context handle (`gss_ctx_id_t`). Normally, this is an input-only parameter and its value is taken from `db2secValidatePassword`. It can also be an output parameter when authentication is done on the client and therefore `db2secValidatePassword` is not called.

*const char *dbname*
The name of the database being connected to. The plug-in can ignore this, or it can return differing authids when the same user connects to different databases.

db2int32 dbnamelen
Length of the dbname.

*const char *usernamespace*
The namespace from which the user ID was obtained.

db2int32 usernamespacelen
Length of the namespace field.

db2int32 usernamespacetype
As specified above. In the current version of DB2, the only supported namespace type is DB2SEC_NAMESPACE_SAM_COMPATIBLE.

Output:

char SystemAuthID[DB2SEC_MAX_AUTHID_LENGTH]
The system authid corresponds to the id of the authenticated user. The size is 255, but DB2 will currently only be able to use up to 30.

*db2int32 *SystemAuthiIDlen*
Length of the `SystemAuthId` returned.

char InitialSessionAuthid[DB2SEC_MAX_AUTHID_LENGTH]
This is the authid used for this connection session. This is usually the same as the `SystemAuthID` but can be different in certain cases such as issuing set session authorization statement. Size is 255, but DB2 will currently only be able to use up to 30.

*db2int32 *InitialSessionAuthidlen*
Length of the `InitialSessionAuthID` returned.

char username[DB2SEC_MAX_USERID_LENGTH]
A username corresponding to the authenticated user and authid. This will only be used for auditing and will be logged in the ″User ID″ field. If the plug-in does not fill in this field, DB2 will copy it from the userid.

*db2int32 *usernamelen*
Length of the user ID returned.

*db2int32 *initsessionidtype*
Session authid type indicating whether or not the `InitialSessionAuthid` is a role or an authid. The plug-in should return the one of the following (defined in `db2secPlugin.h`): `DB2SEC_ID_TYPE_AUTHID` (0) or `DB2SEC_ID_TYPE_ROLE` (1). Currently, DB2 only supports authid (DB2SEC_ID_TYPE_AUTHID).

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- “Security plug-ins” on page 533
- “Security plug-in APIs” on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

db2secDoesAuthIDExist - Check if authentication ID exists

This function will determine if the authid represents an individual user (for instance, whether the function can map the authid to an external user id). This function should return DB2SEC_PLUGIN_OK if it is successful - the authid is valid, DB2SEC_PLUGIN_INVALID_USERORGROUP if it is not valid, or DB2SEC_PLUGIN_USERSTATUSNOTKNOWN if the existence cannot be determined.

C API syntax:

```
SQL_API_RC SQL_API_FN db2secDoesAuthIDExist(
 const char *authid,
 db2int32  authidlen,
 const char *errormsg,
 db2int32  *errormsglen);
```

Input:

*const char *authid*
The authid to validate. This will be upper-cased, with no trailing blanks.

db2int32 authidlen
Length of the authid.

Output:

*char **errormsg*
A pointer to the address of an ASCII string allocated by the plug-in that can be returned in this parameter if the API is not successful.

*db2int32 *errormsglen*
A pointer to an integer that indicates the length of the error message string in *char **errormsg*.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "APIs for user ID/password authentication plug-in" on page 569

GSS-API plug-in APIs

Required APIs and Definitions for GSS-API authentication plug-ins

This topic presents a complete list of GSS-APIs required for the DB2 security plug-in interface. The supported APIs follow these specifications: *Generic Security Service Application Program Interface, Version 2* (IETF RFC2743) and *Generic Security Service API Version 2: C-Bindings* (IETF RFC2744). Before implementing a GSS-API based plug-in, you should have a complete understanding of these specifications.

Table 84. Required APIs and Definitions for GSS-API authentication plug-ins

Name		Description
Client-side APIs		
	`gss_init_sec_context`	Initiate a security context with a peer application
Server-side APIs		
	`gss_accept_sec_context`	Accept a security context initiated by a peer application.
	`gss_display_name`	Convert an internal format name to text.
Common APIs		
	`gss_delete_sec_context`	Delete an established security context.
	`gss_display_status`	Obtain the text error message associated with a GSS-API status code
	`gss_release_buffer`	Delete a buffer.
	`gss_release_cred`	Releases local data structures associated with a GSS-API credential.
	`gss_release_name`	Delete internal format name.
Required definitions		
	`GSS_C_DELEG_FLAG`	Delegation requested.
	`GSS_C_EMPTY_BUFFER`	Signifies that the `gss_buffer_desc` does not contain any data.
	`GSS_C_GSS_CODE`	Indicates a GSS major status code.
	`GSS_C_INDEFINITE`	Indicates that mechanism does not support context expiration.
	`GSS_C_MECH_CODE`	Indicates a GSS minor status code.
	`GSS_C_MUTUAL_FLAG`	Mutual authentication requested.
	`GSS_C_NO_BUFFER`	Signifies that the `gss_buffer_t` variable does not point to a valid `gss_buffer_desc` structure.
	`GSS_C_NO_CHANNEL_BINDINGS`	No communication channel bindings.
	`GSS_C_NO_CONTEXT`	Signifies that the `gss_ctx_id_t` variable does not point to a valid context.
	`GSS_C_NO_CREDENTIAL`	Signifies that `gss_cred_id_t` variable does not point to a valid credential handle.
	`GSS_C_NO_NAME`	Signifies that the `gss_name_t` variable does not point to a valid internal name.
	`GSS_C_NO_OID`	Use default authentication mechanism.
	`GSS_C_NULL_OID_SET`	Use default mechanism.
	`GSS_S_COMPLETE`	API completed successfully.
	`GSS_S_CONTINUE_NEEDED`	Processing is not complete and the API must be called again with the reply token received from the peer.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "Restrictions for GSS-API authentication plug-ins" on page 593

Restrictions for GSS-API authentication plug-ins

The following is a list of restrictions for GSS-API authentication plug-ins.

- The default security mechanism will always be assumed, therefore there is no OID consideration.
- The only GSS services requested in `gss_init_sec_context()` are mutual authentication and delegation. DB2 will always request a ticket for delegation, but will currently not use that ticket to generate a new ticket.
- Only the default context time will be requested.
- Context tokens from `gss_delete_sec_context()` are not sent from the client to the server and vice-versa.
- Anonymity is not supported.
- Channel binding is not supported
- If initial credentials expire, DB2 will not automatically renew them.
- The GSS-API specification stipulates that even if `gss_init_sec_context()` or `gss_accept_sec_context()` fail, either function must return a token to send to the peer. However, because of DRDA limitations, DB2 can only manage to do send a token if `gss_init_sec_context()` fails and generates a token on the first call.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Related reference:

- "Required APIs and Definitions for GSS-API authentication plug-ins" on page 591

Security plug-in API versioning

Since it is possible that future releases of DB2® will need different versions of the security plug-in APIs, DB2 supports version numbering of the APIs. These version numbers will be integers starting with 1 for DB2 UDB Version 8.2. The version number that DB2 passes to the security plug-in APIs will be the highest version number of the API that DB2 can support, and will correspond to a version number of the structure. If the plug-in can support a higher API version, it must return function pointers for the version that DB2 has requested. If the plug-in can only support a lower version of the API, it should fill in function pointers for that lower version. In either case, the security plug-in APIs should return the version number for the API it is supporting in the version field of the functions structure.

For DB2, the version numbers of the security plug-ins will only change when necessary. For example, when there are changes to the parameters of the APIs. Version numbers will not automatically change with DB2 release numbers.

Related concepts:

- "Security plug-ins" on page 533
- "Security plug-in APIs" on page 559

Part 7. General DB2 Application Concepts

Chapter 29. National Language Support

Collating Sequence Overview

The sections that follow describe collating sequences, and how character comparisons are performed.

Collating sequences

The database manager compares character data using a *collating sequence*. This is an ordering for a set of characters that determines whether a particular character sorts higher, lower, or the same as another.

Note: Character string data defined with the FOR BIT DATA attribute, and BLOB data, is sorted using the binary sort sequence.

For example, a collating sequence can be used to indicate that lowercase and uppercase versions of a particular character are to be sorted equally.

The database manager allows databases to be created with custom collating sequences. The following sections help you determine and implement a particular collating sequence for a database.

Each single-byte character in a database is represented internally as a unique number between 0 and 255 (in hexadecimal notation, between X'00' and X'FF').

This number is referred to as the *code point* of the character; the assignment of numbers to characters in a set is collectively called a *code page*. A collating sequence is a mapping between the code point and the desired position of each character in a sorted sequence. The numeric value of the position is called the *weight* of the character in the collating sequence. In the simplest collating sequence, the weights are identical to the code points. This is called the *identity sequence*.

For example, suppose the characters B and b have the code points X'42' and X'62', respectively. If (according to the collating sequence table) they both have a sort weight of X'42' (`B`), they collate the same. If the sort weight for `B` is X'9E', and the sort weight for `b` is X'9D', `b` will be sorted before `B`. The collation sequence table specifies the weight of each character. The table is different from a code page, which specifies the code point of each character.

Consider the following example. The ASCII characters A through Z are represented by X'41' through X'5A'. To describe a collating sequence in which these characters are sorted consecutively (no intervening characters), you can write: X'41', X'42', ... X'59', X'5A'.

The hexadecimal value of a multi-byte character is also used as the weight. For example, suppose the code points for the double-byte characters A and B are X'8260' and X'8261' respectively, then the collation weights for X'82', X'60', and X'61' are used to sort these two characters according to their code points.

The weights in a collating sequence need not be unique. For example, you could give uppercase letters and their lowercase equivalents the same weight.

Specifying a collating sequence can be simplified if the collating sequence provides weights for all 256 code points. The weight of each character can be determined using the code point of the character.

In all cases, DB2 Universal Database™ (DB2 UDB) uses the collation table that was specified at database creation time. If you want the multi-byte characters to be sorted the way that they appear in their code point table, you must specify IDENTITY as the collation sequence when you create the database.

Once a collating sequence is defined, all future character comparisons for that database will be performed with that collating sequence. Except for character data defined as FOR BIT DATA or BLOB data, the collating sequence will be used for all SQL comparisons and ORDER BY clauses, and also in setting up indexes and statistics.

Potential problems can occur in the following cases:

- An application merges sorted data from a database with application data that was sorted using a different collating sequence.
- An application merges sorted data from one database with sorted data from another, but the databases have different collating sequences.
- An application makes assumptions about sorted data that are not true for the relevant collating sequence. For example, numbers collating lower than alphabetics may or may not be true for a particular collating sequence.

A final point to remember is that the results of any sort based on a direct comparison of character code points will only match query results that are ordered using an identity collating sequence.

Related concepts:

- "Character conversion" in the *SQL Reference, Volume 1*
- "Unicode implementation in DB2 Universal Database" in the *Administration Guide: Planning*
- "Character comparisons based on collating sequences" on page 599

Character comparisons based on collating sequences

Once a collating sequence is established for a database with SYSTEM, NLSCHAR, COMPATIBILITY, or user defined collation option, character comparison is performed by comparing the weights of two characters, instead of directly comparing their code point values.

If weights that are not unique are used, characters that are not identical may compare equally. Because of this, string comparison can become a two-phase process:

1. Compare the characters in each string based on their weights.
2. If step 1 yields equality, compare the characters of each string based on their code point values.

If the collating sequence contains 256 unique weights, only the first step is performed. If the collating sequence is the identity sequence, only the second step is performed. In either case, there is a performance benefit. For Unicode databases, if the collation option is SYSTEM or IDENTITY, the collation sequence will be IDENTITY and only the second step is performed.

A Unicode database with the SQL_CS_IDENTITY_16BIT collation option will collate the CHAR or VARCHAR data in the database according to their CESU-8 binary order instead of the UTF-8 binary order. CESU-8 stands for *Compatibility Encoding Scheme for UTF-16: 8-Bit*, as specified in the *Unicode Technical Report #26* available at the Unicode Consortium web site (www.unicode.org). CESU-8 is binary identical to UTF-8, except for the Unicode supplementary characters, that is, those characters that are defined outside the 16-bit Basic Multilingual Plane (BMP or Plane 0). In UTF-8 encoding, a supplementary character is represented by one 4-byte sequence, but the same character in CESU-8 requires two 3-byte sequences. In a Unicode database, character data are stored in UTF-8 and graphic data are stored in UCS-2. For SQL_CS_NONE collation, non-supplementary characters in UTF-8 and UCS-2 have identical binary collation, but supplementary characters in UTF-8 collate differently from the same characters in UCS-2.

For Unicode databases whose collation option is of the UCA (Unicode Collation Algorithm) type, then characters that are not binary identical but semantically equal will compare equally. Because of this, string comparison can become a two-phase process:

1. Compare the characters in each string as per the algorithm specified in the *Unicode Technical Standard #10*, available at the Unicode Technical Consortium web site (www.unicode.org).
2. If step 1 yields equality, compare the characters of each string based on their code point values.

Related concepts:

- "Character conversion" in the *SQL Reference, Volume 1*

Case Independent Comparisons Using the TRANSLATE Function

To perform character comparisons that are independent of case, you can use the TRANSLATE function to select and compare mixed case column data by translating it to uppercase (for purposes of comparison only). Consider the following data:

```
Abel
abels
ABEL
abel
ab
Ab
```

The following SELECT statement:

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

returns

```
ab
Ab
abel
Abel
ABEL
abels
```

You could also specify the following SELECT statement when creating view ″v1″, make all comparisons against the view in uppercase, and request table INSERTs in mixed case:

```
CREATE VIEW v1 AS SELECT TRANSLATE(c1) FROM T1
```

At the database level, you can set the collating sequence as part of the **sqlecrea** - Create Database API. This allows you to decide if ″a″ is processed before ″A″, or if ″A″ is processed after ″a″, or if they are processed with equal weighting. This will make them equal when collating or sorting using the ORDER BY clause. ″A″ will always come before ″a″, because they are equal in every sense. The only basis upon which to sort is the hexadecimal value.

Thus

```
SELECT c1 FROM T1 WHERE c1 LIKE 'ab%'
```

returns

```
ab
abel
abels
```

and

```
SELECT c1 FROM T1 WHERE c1 LIKE 'A%'
```

returns

```
Abel
Ab
ABEL
```

The following statement

```
SELECT c1 FROM T1 ORDER BY c1
```

returns

```
ab
Ab
abel
Abel
ABEL
abels
```

Thus, you may want to consider using the scalar function TRANSLATE(), as well as **sqlecrea**. Note that you can only specify a collating sequence using **sqlecrea**. You cannot specify a collating sequence from the command line processor (CLP).

You can also use the UCASE function as follows, but note that DB2® performs a table scan instead of using an index for the select:

```
SELECT * FROM EMP WHERE UCASE(JOB) = 'NURSE'
```

Related reference:

- "TRANSLATE scalar function" in the *SQL Reference, Volume 1*
- "UCASE or UPPER scalar function" in the *SQL Reference, Volume 1*
- "sqlecrea - Create Database" in the *Administrative API Reference*

Differences Between EBCDIC and ASCII Collating Sequence Sort Orders

The order in which data in a database is sorted depends on the collating sequence defined for the database. For example, suppose that database A uses the EBCDIC code page's default collating sequence and that database B uses the ASCII code page's default collating sequence. Sort orders at these two databases would differ, as shown in the following example:

```
SELECT.....
  ORDER BY COL2

EBCDIC-Based Sort        ASCII-Based Sort

COL2                     COL2
----                     ----
V1G                      7AB
Y2W                      V1G
7AB                      Y2W
```

Figure 64. Example of How a Sort Order in an EBCDIC-Based Sequence Differs from a Sort Order in an ASCII-Based Sequence

Similarly, character comparisons in a database depend on the collating sequence defined for that database. So if database A uses the EBCDIC code page's default collating sequence and database B uses the ASCII code page's default collating sequence, the results of character comparisons at the two databases would differ. The difference is as follows:

```
SELECT.....
  WHERE COL2 > 'TT3'

EBCDIC-Based Results     ASCII-Based Results

COL2                     COL2
----                     ----
TW4                      TW4
X72                      X72
39G
```

Figure 65. Example of How a Comparison of Characters in an EBCDIC-Based Sequence Differs from a Comparison of Characters in an ASCII-Based Sequence

If you are creating a federated database, consider specifying that your collating sequence matches the collating sequence at a data source. This approach will maximize "pushdown" opportunities and possibly increase query performance.

Related concepts:

- "Guidelines for analyzing where a federated query is evaluated" in the *Administration Guide: Performance*

Collating sequence specified when database is created

The collating sequence for a database is specified at database creation time. Once the database has been created, the collating sequence cannot be changed.

The CREATE DATABASE API accepts a data structure called the Database Descriptor Block (SQLEDBDESC). You can define your own collating sequence within this structure.

Note: You can only define your own collating sequence for a single-byte database.

To specify a collating sequence for a database:

- Pass the desired SQLEDBDESC structure, or
- Pass a NULL pointer. The collating sequence of the operating system (based on current country/region code and code page) is used. This is the same as specifying SQLDBCSS equal to SQL_CS_SYSTEM (0).

The SQLEDBDESC structure contains:

SQLDBCSS A 4-byte integer indicating the source of the database collating sequence. Valid values are:

SQL_CS_SYSTEM
The collating sequence of the operating system (based on current country/region code and code page) is used.

SQL_CS_SYSTEM_NLSCHAR
Collating sequence from user using the NLS version of compare routines for character types

SQL_CS_IDENTITY_16BIT
A Unicode database can be created with the SQL_CS_IDENTITY_16BIT collation option. SQL_CS_DENTITY_16BIT differs from the default SQL_CS_NONE collation option in that the CHAR or VARCHAR data in the Unicode database will be

collated using the CESU-8 binary order instead of the UTF-8 binary order. CESU-8 stands for *Compatibility Encoding Scheme for UTF-16: 8-Bit*, as specified in the *Unicode Technical Report #26*, available at the Unicode Consortium web site (www.unicode.org). CESU-8 is binary identical to UTF-8 except for the Unicode supplementary characters, that is, those characters that are defined outside the 16-bit Basic Multilingual Plane (BMP or Plane 0). In UTF-8 encoding, a supplementary character is represented by one 4-byte sequence, but the same character in CESU-8 requires two 3-byte sequences. In a Unicode database, character data are stored in UTF-8, and graphic data are stored in UCS-2. For SQL_CS_NONE collation, non-supplementary characters in UTF-8 and UCS-2 have identical binary collation, but supplementary characters in UTF-8 collate differently from the same characters in UCS-2.
SQL_CS_IDENTITY_16BIT ensures all characters, supplementary and non-supplementary, in a DB2® Unicode databases have the same binary collation.

SQL_CS_UCA_NO
A Unicode database can be created with the SQL_CS_UCA400_NO collation option. SQL_CS_UCA400_NO specifies the UCA (Unicode Collation Algorithm) collation sequence based on the Unicode Standard version 4.00 with normalization implicitly set to on. Details of the UCA can be found in the *Unicode Technical Standard #10*, available at the Unicode Consortium web site (www.unicode.org).

SQL_CS_UCA_LTH
A Unicode database can be created with the SQL_CS_UCA400_LTH collation option. SQL_CS_UCA400_LTH specifies the UCA (Unicode Collation Algorithm) collation sequence based on the Unicode Standard version 4.00, but will sort all Thai characters as per the Royal Thai Dictionary order. Details of the UCA can be found in the *Unicode Technical Standard #10*, available at the Unicode Consortium web site (www.unicode.org).

SQL_CS_USER
The collating sequence is specified by the value in the SQLDBUDC field.

SQL_CS_NONE
The collating sequence is the identity sequence. Strings are compared byte for byte, starting with the first byte, using a simple code point comparison.

Note: These constants are defined in the SQLENV include file.

SQLDBUDC A 256-byte field. The *n*th byte contains the sort weight of the *n*th

character in the code page of the database. If SQLDBCSS is not equal to SQL_CS_USER, this field is ignored.

Related reference:

- "sqlecrea - Create Database" in the *Administrative API Reference*

Sample Collating Sequences

Several sample collating sequences are provided (as include files) to facilitate database creation using the EBCDIC collating sequences instead of the default workstation collating sequence.

The collating sequences in these include files can be specified in the SQLDBUDC field of the SQLEDBDESC structure. They can also be used as models for the construction of other collating sequences.

Include files that contain collating sequences are available for the following host languages:

- C/C++
- COBOL
- FORTRAN

Related reference:

- "Include Files for C and C++" on page 132
- "Include Files for COBOL" on page 176
- "Include Files for FORTRAN" on page 196

Code Pages and Locales

The sections that follow describe code pages, and how code pages and locales are derived.

Derivation of code page values

The *application code page* is derived from the active environment when the database connection is made. If the DB2CODEPAGE registry variable is set, its value is taken as the application code page. However, it is not necessary to set the DB2CODEPAGE registry variable because DB2® will determine the appropriate code page value from the operating system. Setting the DB2CODEPAGE registry variable to incorrect values may cause unpredictable results.

The *database code page* is derived from the value specified (explicitly or by default) at the time the database is created. For example, the following defines how the *active environment* is determined in different operating environments:

UNIX® On UNIX-based operating systems, the active environment is determined from the locale setting, which includes information about language, territory and code set.

Windows® operating systems
For all Windows operating systems, if the DB2CODEPAGE environment variable is not set, the code page is derived from the ANSI code page setting in the Registry.

The *section code page* is derived from the tables used in an SQL statement. If the tables are implicitly or explicitly defined with CCSID ASCII, then the section code page is the same as the database code page. If the tables are defined with CCSID UNICODE, then the section code page is the Unicode code page.

Related reference:

- "Supported territory codes and code pages" in the *Administration Guide: Planning*

Derivation of Locales in Application Programs

Locales are implemented one way on Windows® and another way on UNIX®-based systems. There are two locales on UNIX-based systems:

- The environment locale allows you to specify the language, currency symbol, and so on, that you want to use.
- The program locale contains the current language, currency symbol, and so on, of a program that is running.

On Windows systems, cultural preferences can be set through Regional Settings on the Control Panel. However, there is no environment locale like the one on UNIX-based systems.

When your program is started, it gets a default C locale. It does *not* get a copy of the environment locale. If you set the program locale to any locale other than "C", DB2 Universal Database uses your current program locale to determine the code page and territory settings for your application environment. Otherwise, these values are obtained from the operating system environment. Note that `setlocale()` is not thread-safe, and if you issue `setlocale()` from within your application, the new locale is set for the entire process.

How DB2 Derives Locales

On UNIX®-based systems, the active locale used by DB2® is determined from the LC_CTYPE portion of the locale. For details, see the NLS documentation for your operating system.

- If LC_CTYPE of the program locale has a value other than C, DB2 will use this value to determine the application code page by mapping it to its corresponding code page.
- If LC_CTYPE has a value of C (the C locale), DB2 will set the program locale according to the environment locale, using the `setlocale()` function.
- If LC_CTYPE still has a value of C, DB2 will assume the default of the US English environment, and code page 819 (ISO 8859-1).
- If LC_CTYPE no longer has a value of C, its new value will be used to map to a corresponding code page.

Related reference:

- "Supported territory codes and code pages" in the *Administration Guide: Planning*

Application Considerations

The sections that follow describe considerations that you should be aware of when coding an application.

National Language Support and Application Development Considerations

Constant character strings in static SQL statements are converted at bind time, from the application code page to the database code page, and will be used at execution time in this database code page representation. To avoid such conversions if they are not desired, you can use host variables in place of string constants.

If your program contains constant character strings, you should precompile, bind, compile, and execute the application using the same code page. For a Unicode database, you should use host variables instead of using string constants. The reason for this recommendation is that data conversions by the server can occur in both the bind and the execution phases. This could be a concern if constant character strings are used within the program. These embedded strings are converted at bind time based on the code page which is in effect during the bind phase. Seven-bit ASCII characters are common to all the code pages supported by DB2 Universal Database and will not cause a problem. For non-ASCII characters, users should ensure that the same conversion tables are used by binding and executing with the same active code page.

Any external data obtained by the application will be assumed to be in the application code page. This includes data obtained from a file or from user input. Make sure that data from sources outside the application uses the same code page as the application.

If you use host variables that use graphic data in your C or C++ applications, there are special precompiler, application performance, and application design issues you need to consider. If you deal with EUC code sets in your applications, refer to the applicable topics for guidelines.

When developing an application, you should review the topics that follow this one. Failure to follow the recomentations described in these topics can produce unpredictable conditions. These conditions cannot be detected by the database manager, so no error or warning message will result. For example, a C application contains the following SQL statements operating against a table T1 with one column defined as C1 CHAR(20):

```
     (0)  EXEC SQL CONNECT TO GLOBALDB;
     (1)  EXEC SQL INSERT INTO T1 VALUES ('a-constant');
          strcpy(sqlstmt, "SELECT C1 FROM T1 WHERE C1='a-constant');
     (2)  EXEC SQL PREPARE S1 FROM :sqlstmt;
Where:
    application code page at bind time = x
    application code page at execution time = y
    database code page = z
```

At bind time, '*a-constant*' in statement (1) is converted from code page **x** to code page **z**. This conversion can be noted as (x→z).

At execution time, '*a-constant*' (x→z) is inserted into the table when statement (1) is executed. However, the WHERE clause of statement (2) will be executed with '*a-constant*' (y→z). If the code points in the constant are such that the two conversions (x→z and y→z) yield different results, the SELECT in statement (2) will fail to retrieve the data inserted by statement (1).

Related concepts:

- "Graphic Host Variables in C and C++" on page 143
- "Derivation of code page values" on page 604
- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 614

National Language Support and SQL Statements

The coding of SQL statements is not language dependent. The SQL keywords must be typed as shown, although they may be typed in uppercase, lowercase, or mixed case. The names of database objects, host variables and program labels that occur in an SQL statement must be characters supported by your application code page.

The server does not convert file names. To code a file name, either use the ASCII invariant set, or provide the path in the hexadecimal values that are physically stored in the file system.

In a multi-byte environment, there are four characters which are considered special that do not belong to the invariant character set. These characters are:

- The double-byte percentage and double-byte underscore characters used in LIKE processing.
- The double-byte space character, used for, among other things, blank padding in graphic strings.
- The double-byte substitution character, used as a replacement during code page conversion when no mapping exists between a source code page and a target code page.

The code points for each of these characters, by code page, is as follows:

Table 85. Code Points for Special Double-Byte Characters

Code Page	Double-Byte Percentage	Double-Byte Underscore	Double-Byte Space	Double-Byte Substitution Character
932	X'8193'	X'8151'	X'8140'	X'FCFC'
938	X'8193'	X'8151'	X'8140'	X'FCFC'
942	X'8193'	X'8151'	X'8140'	X'FCFC'
943	X'8193'	X'8151'	X'8140'	X'FCFC'
948	X'8193'	X'8151'	X'8140'	X'FCFC'
949	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
950	X'A248'	X'A1C4'	X'A140'	X'C8FE'
954	X'A1F3'	X'A1B2'	X'A1A1'	X'F4FE'
964	X'A2E8'	X'A2A5'	X'A1A1'	X'FDFE'
970	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
1381	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
1383	X'A3A5'	X'A3DF'	X'A1A1'	X'A1A1'
13488	X'FF05'	X'FF3F'	X'3000'	X'FFFD'
1363	X'A3A5'	X'A3DF'	X'A1A1'	X'A1E0'
1386	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
5039	X'8193'	X'8151'	X'8140'	X'FCFC'

For Unicode databases, the GRAPHIC space is X'0020', which is different from the GRAPHIC space of X'3000' used for euc-Japan and euc-Taiwan databases. Both X'0020' and X'3000' are space characters in the Unicode standard. The difference in the GRAPHIC space code points should be taken into consideration when comparing data from these EUC databases to data from a Unicode database.

Related reference:

- "LIKE predicate" in the *SQL Reference, Volume 1*
- "Extended UNIX Code (EUC) Character Sets" on page 613

Remote routines

When coding routines that will be running remotely, the following considerations apply:

- Data in a routine must be in the code page defined by the PARAMETER CCSID option implicitly or explicitly specified when the routine was created.
- Data passed to or from a routine with a character data type will be code page converted to the section or routine code page as appropriate. Therefore, numeric data and data structures must never be passed with a character type if the client application code page is different from the statement or routine code pages. To avoid character conversion, you can pass data by defining it in binary string format by using a data type of BLOB, or by defining the character data as FOR BIT DATA.

By default, when you invoke routines, they run under a default national language environment, which may not match the database's national language environment. Consequently, using country/region or code-page-specific operations, such as the C `wchar_t` graphic host variables and functions, may not work as you expect. You need to ensure that, if applicable, the correct environment is initialized when you invoke the routine.

Package Name Considerations in Mixed Code Page Environments

Package names are determined when you invoke the PRECOMPILE PROGRAM command or API. By default, they are generated based on the first eight bytes of the application program source file (without the file extension) and are folded to upper case. Optionally, a name can be explicitly defined. Regardless of the origin of a package name, if you are running in an unequal code page environment, the characters for your package names should be in the invariant character set. Otherwise you may experience problems related to the modification of your package name. The database manager will not be able to find the package for the application or a client-side tool will not display the right name for your package.

A package name modification due to character conversion will occur if any of the characters in the package name are not directly mapped to a valid character in the database code page. In such cases, a substitution character replaces the character that is not converted. After such a modification, the package name, when converted back to the application code page, may not match the original package name. An example of a case where this behavior is undesirable is when you use the Control Center to list and work with packages. Package names displayed may not match the expected names.

To avoid conversion problems with package names, ensure that only characters are used which are valid under both the application and database code pages.

Active Code Page for Precompilation and Binding

At precompile/bind time, the precompiler is the executing application. The active code page when the database connection was made prior to the precompile request is used for precompiled statements, and any character data returned in the SQLCA.

Related concepts:

- "Active Code Page for Application Execution" on page 609

Active Code Page for Application Execution

At execution time, the active code page of the user application when a database connection is made is in effect for the duration of the connection. All data is interpreted based on this code page; this includes dynamic SQL statements, user input data, user output data, and character fields in the SQLCA.

Related concepts:

- "Active Code Page for Precompilation and Binding" on page 609

Character conversion between different code pages

Ideally, for optimal performance, your applications should always use the same code page as the statements invoked from the application. However, this is not always practical or possible. The DB2® products provide support for code page conversion that allows your application and database to use different code pages. Characters from one code page must be mapped to the other code page to maintain data integrity.

When code page conversion occurs

Code page conversion can occur in the following situations:

- When a client or application accessing a database is running in a code page that is different from the code page of the statement being invoked:

  You can minimize or eliminate client/server character conversion in some situations. For example, you could:

  - Create a database on Windows NT using code page 850 to match a Windows® client application environment that predominately uses code page 850.

    If a Windows ODBC application is used with the IBM® DB2® ODBC driver in Windows database client, this problem may be alleviated by the use of the TRANSLATEDLL and TRANSLATEOPTION keywords in the `odbc.ini` or `db2cli.ini` file.
  - Create a database on AIX® using code page 850 to match a client application environment that predominately uses code page 850.
  - Avoid specifying the CCSID option when creating tables and the PARAMETER CCSID option when creating routines.
- When a client or application importing a PC/IXF file runs in a code page that is different from the file being imported.

This data conversion will occur on the database client machine before the client accesses the database server. Additional data conversion may take place if the application is running in a code page that is different from the code page of the database (as stated in the previous point).

Data conversion, if any, also depends on how the import utility was called.

- When DB2 Connect is used to access data on a host, AS/400®, or iSeries server. In this case, the data receiver converts the character data. For example, data that is sent to DB2 for MVS/ESA is converted to the appropriate MVS™ coded character set identifier (CCSID) by DB2 for MVS/ESA. The data sent back to the DB2 Connect machine from DB2 for MVS/ESA is converted by DB2 Connect.

Character conversion will **not** occur for:

- File names. You should either use the ASCII invariant set for file names or provide the file name in the hexadecimal values that are physically stored in the file system. Note that if you include a file name as part of an SQL statement, it gets converted as part of the statement conversion.
- Data that is targeted for or comes from a column assigned the FOR BIT DATA attribute, or data used in an SQL operation whose result is FOR BIT or BLOB data. In these cases, the data is treated as a byte stream and no conversion occurs.

  Note: A literal inserted into a column defined as FOR BIT DATA could be converted if that literal was part of an SQL statement that was converted.
- A DB2 product or platform that does not support, or that does not have support installed, for the desired combination of code pages. In this case, an SQLCODE -332 (SQLSTATE 57017) is returned when you try to run your application.

Related concepts:

- "Character conversion" in the *SQL Reference, Volume 1*

Character Substitutions During Code Page Conversions

When your application converts from one code page to another, it is possible that one or more characters are not represented in the target code page. If this occurs, DB2 inserts a *substitution* character into the target string in place of the character that has no representation. The replacement character is then considered a valid part of the string. In situations where a substitution occurs, the SQLWARN10 indicator in the SQLCA is set to 'W'.

Note: Any character conversions resulting from using the WCHARTYPE CONVERT precompiler option will not flag a warning if any substitutions take place.

Related concepts:

- "WCHARTYPE Precompiler Option in C and C++" on page 158

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Supported Code Page Conversions

When data conversion occurs, conversion will take place from a *source code page* to a *target code page*.

The source code page is determined from the source of the data; data from the application has a source code page equal to the application code page, and data from the database has a source code page equal to the database code page.

The determination of target code page is more involved; where the data is to be placed, including rules for intermediate operations, is considered:

- If the data is moved directly from an application into a database, with no intervening operations, the target code page is the database code page.
- If the data is being imported into a database from a PC/IXF file, there are two character conversion steps:
  1. From the PC/IXF file code page (source code page) to the application code page (target code page)
  2. From the application code page (source code page) to the database code page (target code page)

  Exercise caution in situations where two conversion steps might occur. To avoid a possible loss of character data, ensure you follow the supported character conversions. Additionally, within each group, only characters that exist in both the source and target code page have meaningful conversions. Other characters are used as *substitutions* and are only useful for converting from the target code page back to the source code page (and may not necessarily provide meaningless conversions in the two-step conversion process mentioned above). Such problems are avoided if the application code page is the same as the database code page.
- If the data is derived from operations performed on character data, where the source may be any of the application code page, the database code page, FOR BIT DATA, or for BLOB data, data conversion is based on a set of rules. Some or all of the data items may have to be converted to an intermediate result, before the final target code page can be determined.

Note: Code page conversions between multi-byte code pages, for example DBCS and EUC, may result in either an increase or a decrease in the length of the string.

Related concepts:

- "Character conversion" in the *SQL Reference, Volume 1*
- "Character conversion between different code pages" on page 609

Related reference:

- "Supported territory codes and code pages" in the *Administration Guide: Planning*

Code Page Conversion Expansion Factor

When your application successfully completes an attempt to connect to a DB2 database server, you should consider the following fields in the returned SQLCA:

- The second token in the SQLERRMC field (tokens are separated by X'FF') indicates the code page of the database. The ninth token in the SQLERRMC field indicates the code page of the application. Querying the application's code page and comparing it to the database's code page informs the application whether it has established a connection that will undergo character conversions.
- The first and second entries in the SQLERRD array. SQLERRD(1) contains an integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the database code page from the application code page. SQLERRD(2) contains an

integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.

The considerations for graphic string data should not be a factor in unequal code page situations. Each string always has the same number of characters, regardless of whether the data is in the application or the database code page.

Related concepts:

- "Application Development in Unequal Code Page Situations" on page 618

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*

DBCS Character Sets

Each combined single-byte character set (SBCS) or double-byte character set (DBCS) code page allows for both single- and double-byte character code points. This is usually accomplished by reserving a subset of the 256 available code points of a mixed code table for single-byte characters, with the remainder of the code points either undefined, or allocated to the first byte of double-byte code points. These code points are shown in the following table.

Table 86. Mixed Character Set Code Points

Country/Region	Supported Mixed Code Page	Code Points for Single-Byte Characters	Code Points for First Byte of Double-Byte Characters
Japan	932, 943	X'00'-X'7F', X'A1'-X'DF'	X'81'-X'9F', X'E0'-X'FC'
Japan	942	X'00'-X'80', X'A0'-X'DF', X'FD'-X'FF'	X'81'-X'9F', X'E0'-X'FC'
Taiwan	938 (*)	X'00'-X'7E'	X'81'-X'FC'
Taiwan	948 (*)	X'00'-X'80', X'FD', X'FE'	X'81'-X'FC'
Korea	949	X'00'-X'7F'	X'8F'-X'FE'
Taiwan	950	X'00'-X'7E'	X'81'-X'FE'
China	1381	X'00'-X'7F'	X'8C'-X'FE'
Korea	1363	X'00'-X'7F'	X'81'-X'FE'
China	1386	X'00'	X'81'-X'FE'
Note: (*) This is an old code page that is no longer recommended.			

Code points not assigned to either of these categories are not defined, and are processed as single-byte undefined code points.

Within each implied DBCS code table, there are 256 code points available as the second byte for each valid first byte. Second byte values can have any value from

X'40' to X'7E', and from X'80' to X'FE'. Note that in DBCS environments, DB2 does not perform validity checking on individual double-byte characters.

Extended UNIX Code (EUC) Character Sets

Each EUC code page allows for both single-byte character code points, and up to three different sets of multi-byte character code points. This support is accomplished by reserving a subset of the 256 available code points of each implied SBCS code page identifier for single-byte characters. The remainder of the code points is undefined, allocated as an element of a multi-byte character, or allocated as a single-shift introducer of a multi-byte character. These code points are shown in the following tables.

Table 87. Japanese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	n/a	n/a
G3	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	n/a

Table 88. Korean EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

Table 89. Traditional Chinese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	X'A1'-X'FE'
G3	n/a	n/a	n/a	n/a

Table 90. Simplified Chinese EUC Code Points

Group	1st Byte	2nd Byte	3rd Byte	4th Byte
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

Code points not assigned to any of these categories are not defined, and are processed as single-byte undefined code points.

CLI, ODBC, JDBC, and SQLJ Programs in a DBCS Environment

JDBC and SQLJ programs access DB2® using the DB2 CLI/ODBC driver and therefore use the same configuration file (db2cli.ini). The following entries must be added to this configuration file if you run Java™ programs that access DB2 Universal Database in a DBCS environment:

PATCH1 = 65536
Forces the driver to manually insert a ″G″ in front of character literals that are in fact graphic literals. This PATCH1 value should always be set when working in a double-byte environment.

PATCH1 = 64
Forces the driver to NULL terminate graphic output strings. This PATCH1 value is needed by Microsoft® Access in a double-byte environment. If you need to use this PATCH1 value as well, you would add the two values together (64+65536 = 65600) and set PATCH1=65600. See note 2 below for more information about specifying multiple PATCH1 values.

PATCH2 = 7
Forces the driver to map all graphic column data types to char column data type. This PATCH2 value is needed in a double-byte environment.

PATCH2 = 10
Should only be used in an EUC (Extended Unix Code) environment. This PATCH2 value ensures that the CLI driver provides data for character variables (CHAR, VARCHAR, and so on) in the proper format for the JDBC driver. The data in these character types will not be usable in JDBC without this setting.

Notes:

1. Each of these keywords is set in each database specific stanza of the db2cli.ini file. If you want to set them for multiple databases, repeat them for each database stanza in db2cli.ini.
2. To set multiple PATCH1 values, add the individual values and use the sum. To set PATCH1 to both 64 and 65536, set PATCH1=65600 (64+65536). If you already have other PATCH1 values set, replace the existing number with the sum of the existing number and the new PATCH1 values that you want to add.
3. To set multiple PATCH2 values, specify them in a comma delimited string (unlike the PATCH1 option). To set PATCH2 values 1 and 7, set PATCH2=″1,7″

Considerations for Japanese and Traditional Chinese EUC and UCS-2 Code Sets

The sections that follow describe the considerations for Japanese and Traditional Chinese EUC and UCS-2 code sets,

Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations

Extended UNIX® Code (EUC) denotes a set of general encoding rules that can support from one to four character sets in UNIX-based operating environments. The encoding rules are based on the ISO 2022 definition for encoding 7-bit and 8-bit data in which control characters are used to separate some of the character sets. A code set based on EUC conforms to the EUC encoding rules, but also identifies the specific character sets associated with the specific instances. For

example, the IBM®-eucJP code set for Japanese refers to the encoding of the Japanese Industrial Standard characters according to the EUC encoding rules.

Database and client application support for graphic (pure double-byte character) data, while running under EUC code pages with character encoding that is greater than two bytes in length is limited. The DB2 Universal Database products implement strict rules for graphic data that require all characters to be exactly two bytes wide. These rules do not allow many characters from both the Japanese and Traditional Chinese EUC code pages. To overcome this situation, support is provided at both the application level and the database level to represent Japanese and Traditional Chinese EUC graphic data using another encoding scheme.

A database created under either Japanese or Traditional Chinese EUC code pages will actually store and manipulate graphic data using the Unicode UCS-2 code set, a double-byte encoding scheme that is a proper subset of the full Unicode character repertoire. Similarly, an application running under those code pages will send graphic data to the database server as UCS-2 encoded data. With this support, applications running under EUC code pages can access the same types of data as those running under DBCS code pages. The IBM-defined code page identifier associated with UCS-2 is 1200, and the CCSID number for the same code page is 13488. Graphic data in an eucJP or eucTW database uses the CCSID number 13488. In a Unicode database, use CCSID 1200 for GRAPHIC data.

DB2 Universal Database supports the all the Unicode characters that can be encoded using UCS-2, but does not perform any composition, decomposition, or normalization of characters. More information about the Unicode standard can be found at the Unicode Consortium web site, `www.unicode.org`, and from the latest edition of the Unicode Standard book published by Addison Wesley Longman, Inc.

If you are working with applications or databases using these character sets you may need to consider dealing with UCS-2 encoded data. When converting UCS-2 graphic data to the application's EUC code page, there is the possibility of an increase in the length of data. When large amounts of data are being displayed, it may be necessary to allocate buffers, convert, and display the data in a series of fragments.

The following sections discuss how to handle data in this environment. For these sections, the term EUC is used to refer only to Japanese and Traditional Chinese EUC character sets. Note that the discussions do not apply to DB2 Korean or Simplified-Chinese EUC support, because graphic data in these character sets is represented using the EUC encoding.

Related concepts:

- "Code Page Conversion Expansion Factor" on page 611
- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 623

Related reference:

- "Supported territory codes and code pages" in the *Administration Guide: Planning*
- "Extended UNIX Code (EUC) Character Sets" on page 613

Mixed EUC and Double-Byte Client and Database Considerations

The administration of database objects in mixed EUC and double-byte code page environments is complicated by the possible expansion or contraction in the length of object names as a result of conversions between the client and database code page. In particular, many administrative commands and utilities have documented limits to the lengths of character strings that they can take as input or output parameters. These limits are typically enforced at the client, unless documented otherwise. For example, the limit for a table name is 128 bytes. It is possible that a character string that is 128 bytes under a double-byte code page is larger, say 135 bytes, under an EUC code page. This hypothetical 135-byte table name would be considered invalid by such commands as REORGANIZE TABLE if used as an input parameter, despite being valid in the target double-byte database. Similarly, the maximum permitted length of output parameters may be exceeded, after conversion, from the database code page to the application code page. This may cause either a conversion error or output data truncation to occur.

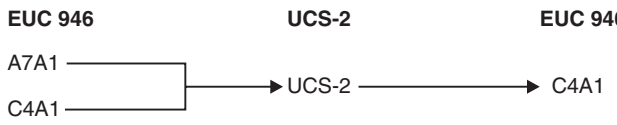
If you expect to use administrative commands and utilities extensively in a mixed EUC and double-byte environment, you should define database objects and their associated data with the possibility of length expansion past the supported limits. Administering an EUC database from a double-byte client imposes fewer restrictions than administering a double-byte database from an EUC client. Double-byte character strings typically are equal or shorter than the corresponding EUC character string. This characteristic will generally lead to fewer problems caused by enforcing the character string length limits.

Note: In the case of SQL statements, validation of input parameters is not conducted until the entire statement has been converted to the database code page. Thus you can use character strings that may be technically longer than allowed when represented in the client code page, but which meet length requirements when represented in the database code page.

Character Conversion Considerations for Traditional Chinese Users

Due to the standards definition for Traditional Chinese, there is a side effect that you may encounter when you convert some characters between double-byte or EUC code pages and UCS-2. There are 189 characters (consisting of 187 radicals and 2 numbers) that share the same UCS-2 code point, when converted, as another character in the code set. When these characters are converted back to double-byte or EUC, they are converted to the code point of the same character's ideograph, with which it shares the same UCS-2 code point, rather then back to the original code point. When displayed, the character appears the same, but has a different code point. Depending on your application's design, you may have to take this behavior into account.

As an example, consider what happens to code point A7A1 in EUC code page 964 when it is converted to UCS-2, then converted back to the original code page, EUC 946:



Thus, the original code points A7A1 and C4A1 end up as code point C4A1 after conversion.

Graphic Data in Japanese or Traditional Chinese EUC Applications

The information that follows describes EUC application development considerations for graphic data, including graphic constants, graphic data in UDFs, stored procedures, DBCLOB files, and collation:

- Graphic constants

  Graphic constants, or literals, are actually classified as mixed character data, as they are part of an SQL statement. Any graphic constants in an SQL statement from a Japanese or Traditional Chinese EUC client are implicitly converted to the graphic encoding by the database server. You can use graphic literals that are composed of EUC encoded characters in your SQL applications. An EUC database server will convert these literals to the graphic database code set, which will be UCS-2. Graphic constants from EUC clients should never contain single-width characters, such as CS0 7-bit ASCII characters or Japanese EUC CS2 (Katakana) characters.

- UDFs

  UDFs are invoked at the database server, and are meant to deal with data encoded in the same code set as the database. In the case of databases running under the Japanese or Traditional Chinese code set, mixed character data is encoded using the EUC code set under which the database is created. Graphic data is encoded using UCS-2. UDFs need to recognize and handle graphic data that is encoded with UCS-2.

  For example, assume that you create a UDF called VARCHAR, and the UDF converts a graphic string to a mixed character string. The VARCHAR function has to convert a graphic string encoded as UCS-2 to an EUC representation if the database is created under the EUC code set.

- Stored procedures

  A stored procedure running under a Japanese or a Traditional Chinese EUC code set must be able to recognize and handle graphic data that is encoded using UCS-2. With these code sets, graphic data that is either received or returned through the stored procedure's input/output SQLDA is encoded using UCS-2.

- DBCLOB files

  The important considerations for DBCLOB files are:

  – The DBCLOB file data is assumed to be in the EUC code page of the application. For EUC DBCLOB files, data is converted to UCS-2 at the client on read, and from UCS-2 at the client on write.

  – The number of bytes read or written at the server is returned in the data length field of the file reference variable. The number of bytes is based on the number of UCS-2 encoded characters that are either read from or written to the file. The number of bytes actually read from or written to the file may be larger than the server writes in the data length field.

- Collation

  Graphic data is sorted in binary sequence. Mixed data is sorted in the collating sequence of the database applied on each byte. Because of the possible difference in the ordering of characters in an EUC code set and a DBCS code set for the same country/region, different results may be obtained when the same data is sorted in an EUC database and in a DBCS database.

Related reference:

- "GRAPHIC scalar function" in the *SQL Reference, Volume 1*
- "SELECT statement" in the *SQL Reference, Volume 2*
- "Graphic strings" in the *SQL Reference, Volume 1*

Application Development in Unequal Code Page Situations

Depending on the character encoding schemes used by the application code page and the database code page, there may or may not be a change in the length of a string as it is converted from the source code page to the target code page. A change in length is usually associated with conversions between multi-byte code pages with different encoding schemes, for example DBCS and EUC.

A possible increase in length is usually more serious than a possible decrease in length, because an over-allocation of memory is less problematic than an under-allocation. Application considerations for sending or retrieving data depending on where the possible expansion may occur need to be dealt with separately. It is also important to note the differences between a *best-case* and *worst-case* situation when an expansion or contraction in length is indicated. Positive values, indicating a possible expansion, will give the *worst-case* multiplying factor. For example, a value of `2` for the SQLERRD(1) or SQLERRD(2) field means that a maximum of twice the string length of storage will be required to handle the data after conversion. This is a *worst-case* indicator. In this example, *best-case* would be that after conversion the length remains the same.

Negative values for SQLERRD(1) or SQLERRD(2), indicating a possible contraction, also provide the *worst-case* expansion factor. For example, a value of `-1` means that the maximum storage required is equal to the string length prior to conversion. It is indeed possible that less storage may be required, but practically this is of little use unless the receiving application knows in advance how the source data is structured.

To ensure that you always have sufficient storage allocated to cover the maximum possible expansion after character conversion, you should allocate storage equal to the value `max_target_length` obtained from the following calculation:

1. Determine the expansion factor for the data.

   For data transfer from the application to the database:

   ```
   expansion_factor = ABS[SQLERRD(1)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   For data transfer from the database to the application:

   ```
   expansion_factor = ABS[SQLERRD(2)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   In the above calculations, `ABS` refers to the absolute value.

   The check for `expansion_factor = 0` is necessary because some DB2 Universal Database products return 0 in SQLERRD(1) and SQLERRD(2). These servers do not support code page conversions that result in the expansion or shrinkage of data; this is represented by an expansion factor of 1.

2. Intermediate length calculation.

   ```
   temp_target_length = actual_source_length * expansion_factor
   ```

3. Determine the maximum length for target data type.

Target data type	Maximum length of type (type_maximum_length)
CHAR	254
VARCHAR	32 672
LONG VARCHAR	32 700
CLOB	2 147 483 647

4. Determine the maximum target length.

```
1  if temp_target_length < actual_source_length
      max_target_length = type_maximum_length
   else
2  if temp_target_length > type_maximum_length
      max_target_length = type_maximum_length
   else
3  max_target_length = temp_target_length
```

All the above checks are required to allow for overflow, which may occur during the length calculation. The specific checks are:

1 Numeric™ overflow occurs during the calculation of `temp_target_length` in step 2.

If the result of multiplying two positive values together is greater than the maximum value for the data type, the result *wraps around* and is returned as a value less than the larger of the two values.

For example, the maximum value of a 2-byte signed integer (which is used for the length of non-CLOB data types) is 32 767. If the `actual_source_length` is 25 000 and the expansion factor is 2, `temp_target_length` is theoretically 50 000. This value is too large for the 2-byte signed integer so it gets wrapped around and is returned as -15 536.

For the CLOB data type, a 4-byte signed integer is used for the length. The maximum value of a 4-byte signed integer is 2 147 483 647.

2 `temp_target_length` is too large for the data type.

The length of a data type cannot exceed the values listed in step 3.

If the conversion requires more space than is available in the data type, it may be possible to use a larger data type to hold the result. For example, if a CHAR(250) value requires 500 bytes to hold the converted string, it will not fit into a CHAR value because the maximum length is 254 bytes. However, it may be possible to use a VARCHAR(500) to hold the result after conversion. See the topic on code page conversion string-length overflow in mixed code set environments for more information about what happens when converted data exceeds the limit for a data type.

3 `temp_target_length` is the correct length for the result.

Using the SQLERRD(1) and SQLERRD(2) values returned when connecting to the database and the above calculations, you can determine whether the length of a string will possibly increase or decrease as a result of character conversion. In general, a value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. (Note that values of '0' will only come from down-level DB2 Universal Database products. Also, these values are undefined for other database server products. The

following table lists values to expect for various application code page and database code page combinations when using DB2 Universal Database.

Table 91. SQLCA.SQLERRD Settings on CONNECT

Application Code Page	Database Code Page	SQLERRD(1)	SQLERRD(2)
SBCS	SBCS	+1	+1
DBCS	DBCS	+1	+1
eucJP	eucJP	+1	+1
eucJP	DBCS	-1	+2
DBCS	eucJP	+2	-1
eucTW	eucTW	+1	+1
eucTW	DBCS	-1	+2
DBCS	eucTW	+2	-1
eucKR	eucKR	+1	+1
eucKR	DBCS	+1	+1
DBCS	eucKR	+1	+1
eucCN	eucCN	+1	+1
eucCN	DBCS	+1	+1
DBCS	eucCN	+1	+1

If the SQLERRD(1) or SQLERRD(2) values indicate an expansion at either the database server or the application client, you should consider the following:

- Expansion at the database server

  If the SQLERRD(1) entry indicates an expansion at the database server, your application must consider the possibility that length-dependent character data that is valid at the client will not be valid at the database server after it is converted. For example, DB2 products require that column names be no more than 128 bytes in length. It is possible that a character string that is 128 bytes in length encoded under a DBCS code page expands past the 128-byte limit when it is converted to an EUC code page. This possibility means that there may be activities that are valid when the application code page and the database code page are equal, and invalid when they are different. Exercise caution when you design EUC and DBCS databases for unequal code page situations.
- Expansion at the application

  If the SQLERRD(2) entry indicates an expansion at the client application, your application must consider the possibility that length-dependent character data will expand in length after being converted. For example, a row with a CHAR(128) column is retrieved. When the database and application code pages are equal, the length of the data returned is 128 bytes. However, in an unequal code page situation, 128 bytes of data encoded under a DBCS code page may expand past 128 bytes when converted to an EUC code page. Thus, additional storage may have to allocated to retrieve the complete string.

Related concepts:

- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 623

Client-Based Parameter Validation in a Mixed Code Set Environment

An important side effect of potential character data expansion or contraction between the client and server involves the validation of data passed between the client application and the database server. In an unequal code page situation, it is possible that data determined to be valid at the client is actually invalid at the database server after code page conversion. Conversely, data that is invalid at the client may be valid at the database server after conversion.

Any end-user application or API library has the potential of not being able to handle all possibilities in an unequal code page situation. In addition, while some parameter validation, such as string length, is performed at the client for commands and APIs, the tokens within SQL statements are not verified until they have been converted to the database's code page. This verification can lead to situations where it is possible to use an SQL statement in an unequal code page environment to access a database object, such as a table, but it will not be possible to access the same object using a particular command or API.

Consider an application that returns data contained in a table provided by an end-user, and checks that the table name is not greater than 128 bytes long. Now consider the following scenarios for this application:

1. A DBCS database is created. From a DBCS client, a table (t1) is created with a table name which is 128 bytes long. The table name includes several characters which would be greater than two bytes in length if the string is converted to EUC, resulting in the EUC representation of the table name being a total of 131 bytes in length. Because there is no expansion for DBCS to DBCS connections, the table name is 128 bytes in the database environment, and the CREATE TABLE is successful.
2. An EUC client connects to the DBCS database. It creates a table (t2) with a table name that is 120 bytes long when encoded as EUC, and 100 bytes long when converted to DBCS. The table name in the DBCS database is 100 bytes. The CREATE TABLE is successful.
3. The EUC client creates a table (t3) with a table name that is 64 EUC characters in length (131 bytes). When this name is converted to DBCS, its length shrinks to the 128-byte limit. The CREATE TABLE is successful.
4. The EUC client invokes the application against the each of the tables (t1, t2, and t3) in the DBCS database, which results in:

Table	Result
t1	The application considers the table name invalid because it is 131 bytes long.
t2	Displays correct results
t3	The application considers the table name invalid because it is 131 bytes long.

5. The EUC client is used to query the DBCS database from the CLP. Although the table name is 131 bytes long on the client, the queries are successful because the table name is 128 bytes long at the server.

DESCRIBE Statement in Mixed Code Set Environments

A DESCRIBE performed against an EUC database will return information about mixed character and GRAPHIC columns based on the definition of these columns in the database. This information is based on code page of the server before it is converted to the client's code page.

When you perform a DESCRIBE against a select list item that is resolved in the application context (for example `VALUES SUBSTR(?,1,2)`) then, for any character or graphic data involved, you should evaluate the returned SQLLEN value along with the returned code page. If the returned code page is the same as the application code page, there is no expansion. If the returned code page is the same as the database code page, expansion is possible. Select list items that are FOR BIT DATA (code page 0) or in the application code page are not converted when returned to the application, therefore there is no expansion or contraction of the reported length.

Considerations are different for an EUC application accessing a DBCS database as compared to a DBCS application accessing an EUC database:

- EUC application accessing a DBCS database

  If your application's code page is an EUC code page, and it issues a DESCRIBE against a database with a DBCS code page, the information returned for CHAR and GRAPHIC columns is returned in the database context. For example, a CHAR(5) column returned as part of a DESCRIBE has a value of five for the SQLLEN field. In the case of non-EUC data, you allocate five bytes of storage when you fetch the data from this column. With EUC data, this may not be the case. When the code page conversion from DBCS to EUC takes place, there may be an increase in the length of the data due to the different encoding used for characters for CHAR columns. For example, with the Traditional Chinese character set, the maximum increase is double. That is, the maximum character length in the DBCS encoding is two bytes, which may increase to a maximum character length of four bytes in EUC. For the Japanese code set, the maximum increase is also double. Note, however, that while the maximum character length in Japanese DBCS is two bytes, it may increase to a maximum character length in Japanese EUC of three bytes. Although this increase appears to be only by a factor of 1.5, the single-byte Katakana characters in Japanese DBCS are only one byte in length, while they are two bytes in length in Japanese EUC.

  Possible changes in data length as a result of character conversions apply only to mixed character data. Graphic character data encoding is always the same length, two bytes, regardless of the encoding scheme. To avoid losing the data, you need to evaluate whether an unequal code page situation exists, and whether or not it is between an EUC application and a DBCS database. You can determine the database code page and the application code page from tokens in the SQLCA returned from a CONNECT statement. If such a situation exists, your application needs to allocate additional storage for mixed character data based on the maximum expansion factor for that encoding scheme.

- DBCS application accessing an EUC database

  If your application code page is a DBCS code page and issues a DESCRIBE against an EUC database, the situation is similar to that in in which an EUC application accesses a DBCS database. However, in this situation your application may require less storage than is indicated by the value of the SQLLEN field. The worst case in this situation is that all of the data is single-byte or double-byte under EUC, meaning that exactly SQLLEN bytes are

required under the DBCS encoding scheme. In any other situation, less than SQLLEN bytes are required because a maximum of two bytes is required to store any EUC character.

Related concepts:

- "Derivation of code page values" on page 604
- "Code Page Conversion Expansion Factor" on page 611
- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 623

Related reference:

- "DESCRIBE statement" in the *SQL Reference, Volume 2*

Fixed-Length and Variable-Length Data in Mixed Code Set Environments

Due to the possible change in length of strings when conversions occur between DBCS and EUC code pages, you should consider not using fixed-length data types. Depending on whether you require blank padding, you should consider changing the SQLTYPE from a fixed-length character string to a variable-length character string after performing the DESCRIBE. For example, if an EUC to DBCS connection is informed of a maximum expansion factor of two for a CHAR(5) column, the application should allocate ten bytes.

If the SQLTYPE is fixed-length, the EUC application will receive the column as an EUC data stream converted from the DBCS data (which itself may have up to five bytes of trailing blank pads) with further blank padding if the code page conversion does not cause the data element to grow to its maximum size. If the SQLTYPE is variable-length, the original meaning of the content of the CHAR(5) column is preserved, however, the source five bytes may have a target of between five and ten bytes. Similarly, in the case of possible data shrinkage (DBCS application and EUC database), you should consider working with variable-length data types.

An alternative to either allocating extra space or promoting the data type is to select the data in fragments. For example, to select the same VARCHAR(3000), which may be up to 6 000 bytes in length after the conversion, you could perform two selects, `SUBSTR(VC3000, 1, LENGTH(VC3000)/2)` and `SUBSTR(VC3000, (LENGTH(VC3000)/2)+1)`, separately into 2 VARCHAR(3000) application areas. This method is the only possible solution when the data type is no longer promotable. For example, a CLOB encoded in the Japanese DBCS code page with the maximum length of 2 gigabytes is possibly up to twice that size when encoded in the Japanese EUC code page. This means that the data will have to be broken up into fragments, because there is no support for a data type in excess of 2 gigabytes in length.

Code Page Conversion String-Length Overflow in Mixed Code Set Environments

In EUC and DBCS unequal code page environments, situations may occur after conversion takes place, when there is not enough space allocated in a column to accommodate the entire string. In this case, the maximum expansion will be twice the length of the string in bytes. In cases where expansion does exceed the capacity of the column, SQLCODE -334 (SQLSTATE 22524) is returned.

This leads to situations that may not be immediately obvious or previously considered as follows:

- An SQL statement may be no longer than 32 765 bytes in length. If the statement is complex enough or uses enough constants or database object names that may be subject to expansion upon conversion, this limit may be reached earlier than expected.
- SQL identifiers are allowed to expand on conversion up to their maximum length, which is eight bytes for short identifiers and 128 bytes for long identifiers.
- Host language identifiers are allowed to expand on conversion up to their maximum length, which is 255 bytes.
- When the character fields in the SQLCA structure are converted, they are allowed to expand to no more than their maximum defined length.

When you design applications for mixed code set environments, you should refer to the appropriate documentation if you have any of the following situations:

- Corresponding string columns in full selects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE statement
- Arguments of the scalar function COALESCE (and VALUE)
- Expression values of the IN list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause

In these situations, conversions may occur according to the application code page instead of the database code page.

Other situations that you need to consider are those in which the character conversion results in a string length beyond the limit for the data type, and code page conversions in stored procedures:

- Character conversion past a data type limit

  In EUC and DBCS unequal code page environments, situations may occur after conversion takes place in which the length of the mixed character or graphic string exceeds the maximum length allowed for that data type. If the length of the string, after expansion, exceeds the limit of the data type, type promotion does not occur. Instead, an error message is returned indicating that the maximum allowed expansion length has been exceeded. This situation is more likely to occur while evaluating predicates than inserts. With inserts, the column width is more readily known by the application, and the maximum expansion factor can be readily taken into account. In many cases, this side effect of character conversion can be avoided by casting the value to an associated data type with a longer maximum length. For example, the maximum length of a CHAR value is 254 bytes, while the maximum length of a VARCHAR is 32 672 bytes. In cases where expansion does exceed the maximum length of the data type, SQLCODE -334 (SQLSTATE 22524) is returned.
- Code page conversion in a stored procedure

  Mixed character or graphic data specified in host variables and SQLDAs in `sqleproc()` or SQL CALL invocations are converted in situations where the application and database code pages are different. In cases where string length expansion occurs as a result of conversion, you receive an SQLCODE -334 (SQLSTATE 22524) if there is not enough space allocated to handle the expansion. Thus you must be sure to provide enough space for potentially

expanding strings when developing stored procedures. You should use variable-length data types with enough space allocated to allow for expansion.

Related reference:

- "COALESCE scalar function" in the *SQL Reference, Volume 1*
- "VALUE scalar function" in the *SQL Reference, Volume 1*
- "Fullselect" in the *SQL Reference, Volume 1*
- "VALUES statement" in the *SQL Reference, Volume 2*
- "CASE statement" in the *SQL Reference, Volume 2*
- "Predicates" in the *SQL Reference, Volume 1*

Applications Connected to Unicode Databases

Applications from any code page environment can connect to a Unicode database. For applications that connect to a Unicode database, the database manager converts character string data between the application code page and the database code page (UTF-8). When DB2 converts characters from a code page to UTF-8, the total number of bytes that represent the characters may expand or shrink, depending on the code page and the code points of the characters. 7-bit ASCII remains invariant in UTF-8, and each ASCII character requires one byte. Non-ASCII characters become more than one byte each. For more information about UTF-8 conversions, refer to the Unicode standard documents.

Note: The information that applies to applications in mixed code sets also applies to applications that connect to Unicode databases.

For a Unicode database, GRAPHIC data is in UCS-2 big-endian order. If you use the command line processor to retrieve graphic data, the graphic characters are also converted to the client code page. This conversion allows the command line processor to display graphic characters in the current font. Data loss may occur whenever the database manager converts UCS-2 characters to a client code page. Characters that the database manager cannot convert to a valid character in the client code page are replaced with the default substitution character in that code page.

Starting with DB2® Version 8, the database manager checks the code page setting of the client, and performs all required conversions for UCS-2 GRAPHIC data. For example, if a non-Unicode application sends GRAPHIC data, DB2 converts the GRAPHIC data to UCS-2 before the data is stored in a UCS-2 database. Conversely, if a non-Unicode application requests GRAPHIC data from a UCS-2 database, DB2 converts the GRAPHIC data to the code page of the application before the application can access the data.

Note: The following restrictions apply:

- When the DB2 Load, Import, or Export utilities are working with a DBCLOB file, the database manager does not check the code page of the client.
- When GRAPHIC data is retrieved from a UCS-2 database to a non-SBCS, non-EUC, or non-Unicode application, DB2 substitutes an ASCII blank character (U+0020) for each blank that is padded to the UCS-2 GRAPHIC column. The substitution is performed because pure DBCS code pages have no equivalent to the UCS-2 blank.

- When DATE, TIME, and TIMESTAMP data is retrieved from a UCS-2 database as a GRAPHIC data type to a non-SBCS, non-EUC, or non-Unicode application, DB2 converts these data types to the substitution character. The substitution is performed because the UCS-2 data types contain SBCS characters that have no equivalent in pure DBCS code pages.

Before Version 8, DB2 did not perform any automatic conversion of UCS-2 GRAPHIC data. Non-Unicode applications had to perform the necessary conversions to and from Unicode themselves, or set the WCHARTYPE CONVERT option and use `wchar_t`. If a Version 7 client connects to a DB2 Version 8 server, the database manager, by default, does *not* perform data conversion for UCS-2 GRAPHIC data. If you want to override this default behaviour, you can set the DB2GRAPHICUNICODESERVER registry variable to `OFF`.

For applications that connect to DBCS databases, GRAPHIC data is converted between the application DBCS code page and the database DBCS code page.

Related concepts:

- "Unicode handling of data types" in the *Administration Guide: Planning*
- "String comparisons in a Unicode database" in the *Administration Guide: Planning*
- "Graphic Host Variables in C and C++" on page 143
- "Package Name Considerations in Mixed Code Page Environments" on page 608
- "Mixed EUC and Double-Byte Client and Database Considerations" on page 616
- "Client-Based Parameter Validation in a Mixed Code Set Environment" on page 621
- "DESCRIBE Statement in Mixed Code Set Environments" on page 622
- "Fixed-Length and Variable-Length Data in Mixed Code Set Environments" on page 623
- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 623

Chapter 30. Managing Transactions

Remote Unit of Work

A unit of work is a single logical transaction. It consists of a sequence of SQL statements in which either all of the operations are successfully performed, or the sequence as a whole is considered unsuccessful.

A remote unit of work lets a user or application program read or update data at one location per unit of work. It supports access to one database within a unit of work. While an application program can access several remote databases, it can only access one database within a unit of work.

A remote unit of work has the following characteristics:
- Multiple requests per unit of work are supported.
- Multiple cursors per unit of work are supported.
- Each unit of work can access only one database.
- The application program either commits or rolls back the unit of work. In certain error conditions, the server may roll back the unit of work.

Multisite Update Considerations

The sections that follow describe multisite updates, and how to develop applications that perform multisite updates.

Multisite Update

Multisite update, also known as *distributed unit of work* (DUOW) and *two-phase commit*, is a function that enables your applications to update data in multiple remote database servers with guaranteed integrity. A good example of a multisite update is a banking transaction that involves the transfer of money from one

account to another in a different database server. In such a transaction it is critical that updates that implement debit operation on one account do not get committed unless the updates required to process credit to the other account are committed as well. The multisite update considerations apply when data representing these accounts is managed by two different database servers.

You can use multisite update to read and update multiple DB2 Universal Database databases within a unit of work. If you have installed DB2® Connect or use the DB2 Connect™ capability provided with DB2 Universal Database™ Enterprise Edition, you can also use multisite update with host, AS/400®, or iSeries database servers such as DB2 Universal Database for z/OS and OS/390 and DB2 UDB for AS/400. Certain restrictions apply when you use DB2 Connect in a multisite update with other database servers.

A transaction manager coordinates the commit among multiple databases. If you use a transaction processing (TP) monitor environment such as TxSeries CICS®, the TP monitor uses its own transaction manager. Otherwise, the transaction manager supplied with DB2 is used. DB2 Universal Database for UNIX®, and Windows® 32-bit operating systems is an XA (extended architecture) compliant resource manager. Host and iSeries database servers that you access with DB2 Connect are XA compliant resource managers. Also note that the DB2 Universal Database transaction manager is*not* an XA compliant transaction manager, meaning the transaction manager can only coordinate DB2 databases.

Related concepts:

- "X/Open distributed transaction processing model" in the *Administration Guide: Planning*
- "Multisite Updates" in the *DB2 Connect User's Guide*

When to Use Multisite Update

Multisite update is most useful when you want to work with two or more databases and maintain data integrity. For example, if each branch of a bank has its own database, a money transfer application could do the following:

1. Connect to the sender's database.
2. Read the sender's account balance and verify that enough money is present.
3. Reduce the sender's account balance by the transfer amount.
4. Connect to the recipient's database
5. Increase the recipient's account balance by the transfer amount.
6. Commit the databases.

By doing the transfer of funds within one unit of work, you ensure that either both databases are updated or neither database is updated.

SQL Statements in Multisite Update Applications

The following table shows how you code SQL statements for multisite update. The left column shows SQL statements that do not use multisite update; the right column shows similar statements with multisite update.

Table 92. RUOW and Multisite Update SQL Statements

RUOW Statements	Multisite Update Statements

CONNECT TO D1
  SELECT
  UPDATE
  COMMIT
 	
CONNECT TO D2
  INSERT
  COMMIT
 	
CONNECT TO D1
  SELECT
  COMMIT
CONNECT RESET
	CONNECT TO D1
	  SELECT
	  UPDATE
	
	CONNECT TO D2
	  INSERT
	  RELEASE CURRENT
	
	SET CONNECTION D1
	  SELECT
	  RELEASE D1
	  COMMIT

The SQL statements in the left column access only one database for each unit of work. This is a remote unit of work (RUOW) application.

The SQL statements in the right column access more than one database within a unit of work. This is a multisite update application.

Some SQL statements are coded and interpreted differently in a multisite update application:

- The current unit of work does not need to be committed or rolled back before you connect to another database.
- When you connect to another database, the current connection is not disconnected. Instead, it is put into a *dormant* state. If the CONNECT statement fails, the current connection is not affected.
- You cannot connect with the USER/USING clause if a current or dormant connection to the database already exists.
- You can use the SET CONNECTION statement to change a dormant connection to the current connection.

  You can also accomplish the same thing by issuing a CONNECT statement to the dormant database. This method is not allowed if you set SQLRULES to STD. You can set the value of SQLRULES using a precompiler option or the SET CLIENT command or API. The default value of SQLRULES (DB2) allows you to switch connections using the CONNECT statement.
- In a select, the cursor position is not affected if you switch to another database, then back to the original database.
- The CONNECT RESET statement does not disconnect the current connection and does not implicitly commit the current unit of work. Instead, this statement is equivalent to explicitly connecting to the default database (if one has been defined). If an implicit connection is not defined, SQLCODE -1024 (SQLSTATE 08003) is returned.
- You can use the RELEASE statement to mark a connection for disconnection at the next COMMIT. The RELEASE CURRENT statement applies to the current connection, the RELEASE *connection* applies to the named connection, and the RELEASE ALL statement applies to all connections.

  A connection that is marked for release can still be used until it is dropped at the next COMMIT statement. A rollback does not drop the connection; this behavior allows a retry with the connections still in place. Use the DISCONNECT statement (or precompiler option) to drop connections after a commit or rollback.

- The COMMIT statement commits all databases in the unit of work (current or dormant).
- The ROLLBACK statement rolls back all databases in the unit of work, and closes held cursors for all databases whether or not they are accessed in the unit of work.
- All connections (including dormant connections and connections marked for release) are disconnected when the application process terminates.
- Upon any successful connection (including a CONNECT statement with no options, which only queries the current connection) a number will be returned in the SQLERRD(3) and SQLERRD(4) fields of the SQLCA.

  The SQLERRD(3) field returns information on whether the database connected is currently updatable in a unit of work. Its possible values are:

  1 Updatable.
  2 Read-only.

  The SQLERRD(4) field returns the following information on the current characteristics of the connection:

  0 Not applicable. This state is only possible if running from a down-level client that uses one-phase commit and is an updater.

  1 One-phase commit.

  2 One-phase commit (read-only). This state is only applicable to host, AS/400®, or iSeries database servers that you access with DB2® Connect *without* starting the DB2 Connect™ sync point manager.

  3 Two-phase commit.

  If you are writing tools or utilities, you may want to issue a message to your users if the connection is read-only.

Precompilation of Multisite Update Applications

When you precompile a multisite update application, you should set the CLP connection to a type 1 connection; otherwise, you will receive an SQLCODE 30090 (SQLSTATE 25000) when you attempt to precompile your application. The following precompiler options are used when you precompile an application that uses multisite updates:

CONNECT (1 | 2)
Specify CONNECT 2 to indicate that this application uses the SQL syntax for multisite update applications. The default, CONNECT 1, means that the normal (RUOW) rules for SQL syntax apply to the application.

SYNCPOINT (ONEPHASE | TWOPHASE | NONE)
If you specify SYNCPOINT TWOPHASE and DB2® coordinates the transaction, DB2 requires a database to maintain the transaction state information. When you deploy your application, you must define this database by configuring the database manager configuration parameter *tm_database*.

SQLRULES (DB2 | STD)
Specifies whether DB2 rules or standard (STD) rules based on ISO/ANSI SQL92 should be used in multisite update applications. DB2 rules allow you to issue a CONNECT statement to a dormant database; STD rules do not allow this.

DISCONNECT (EXPLICIT | CONDITIONAL | AUTOMATIC)
Specifies which database connections are disconnected at COMMIT: only

databases that are marked for release with a RELEASE statement (EXPLICIT), all databases that have no open WITH HOLD cursors (CONDITIONAL), or all connections (AUTOMATIC).

Multisite update precompiler options become effective when the first database connection is made. You can use the SET CLIENT API to supersede connection settings when there are no existing connections (before any connection is established or after all connections are disconnected). You can use the QUERY CLIENT API to query the current connection settings of the application process.

The binder fails if an object referenced in your application program does not exist. There are three possible ways to deal with multisite update applications:

- You can split the application into several files, each of which accesses only one database. You then prep and bind each file against the one database that it accesses.
- You can ensure that each table exists in each database. For example, the branches of a bank might have databases whose tables are identical (except for the data).
- You can use only dynamic SQL.

Related concepts:

- "SQL Statements in Multisite Update Applications" on page 628

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*
- "sqlesetc - Set Client" in the *Administrative API Reference*
- "sqleqryi - Query Client Information" in the *Administrative API Reference*
- "PRECOMPILE Command" in the *Command Reference*

Configuration Parameter Considerations for Multisite Update Applications

The following configuration parameters affect applications which perform multisite updates. With the exception of *locktimeout*, the configuration parameters are database manager configuration parameters. *locktimeout* is a database configuration parameter.

tm_database
: Specifies which database will act as a transaction manager for two-phase commit transactions.

resync_interval
: Specifies the number of seconds that the system waits between attempts to try to resynchronize an indoubt transaction. (An indoubt transaction is a transaction that successfully completes the first phase of a two-phase commit but fails during the second phase.)

locktimeout
: Specifies the number of seconds before a lock wait will time-out and roll back the current transaction for a given database. The application must issue an explicit ROLLBACK to roll back all databases that participate in the multisite update. *locktimeout* is a database configuration parameter.

tp_mon_name
Specifies the name of the TP monitor, if any.

spm_resync_agent_limit
Specifies the number of simultaneous agents that can perform resync operations with the host, AS/400®, or iSeries server using SNA.

spm_name

- If the sync point manager is being used with a TCP/IP two-phase commit connection, the *spm_name* must be an unique identifier within the network. When you create a DB2® instance, DB2 derives the default value of *spm_name* from the TCP/IP hostname. You may modify this value if it is not acceptable in your environment. For TCP/IP connectivity with host database servers, the default value should be acceptable. For SNA connections to host, AS/400, or iSeries database servers, this value must match an SNA LU profile defined within your SNA product.
- If the sync point manager is being used with an SNA two-phase commit connection, the sync point manager name must be set to the LU_NAME that is used for two-phase commit.
- If the sync point manager is being used for both TCP/IP and SNA, the LU_NAME that is used for two-phase commit must be used.

Note: Multisite updates in an environment with host, AS/400, or iSeries database servers may require the sync point manager.

spm_log_size
The number of 4 kilobyte pages of each primary and secondary log file used by the sync point manager to record information on connections, status of current connections, and so on.

Additional considerations exist if your application performs multisite updates that are coordinated by an XA transaction manager with connections to a host, AS/400, or iSeries database.

Related concepts:

- "Multisite Updates" in the *DB2 Connect User's Guide*
- "Multisite update and sync point manager" in the *DB2 Connect User's Guide*

Related tasks:

- "Enabling Multisite Updates using the Control Center" in the *DB2 Connect User's Guide*

Related reference:

- "spm_log_path - Sync point manager log file path configuration parameter" in the *Administration Guide: Performance*
- "resync_interval - Transaction resync interval configuration parameter" in the *Administration Guide: Performance*
- "tm_database - Transaction manager database name configuration parameter" in the *Administration Guide: Performance*
- "tp_mon_name - Transaction processor monitor name configuration parameter" in the *Administration Guide: Performance*
- "locktimeout - Lock timeout configuration parameter" in the *Administration Guide: Performance*

- "spm_name - Sync point manager name configuration parameter" in the *Administration Guide: Performance*
- "spm_log_file_sz - Sync point manager log file size configuration parameter" in the *Administration Guide: Performance*
- "spm_max_resync - Sync point manager resync agent limit configuration parameter" in the *Administration Guide: Performance*

Accessing Host, AS/400, or iSeries Servers

Procedure:

If you want to develop applications that can access (or update) different database systems, you should:

1. Use SQL statements and precompile/bind options that are supported on all of the database systems that your applications will access. For example, stored procedures are not supported on all platforms.

   For IBM products, see the SQL documentation *before* you start coding.
2. Where possible, have your applications check the SQLSTATE rather than the SQLCODE.

   If your applications will use DB2 Connect and you want to use SQLCODE values, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.
3. Test your application with the host, AS/400, or iSeries databases (such as DB2 Universal Database for z/OS and OS/390, OS/400, or DB2 Server for VSE & VM) that you intend to support.

Related concepts:

- "Applications in Host or iSeries Environments" on page 691

Concurrent Transactions

The sections that follow describe concurrent transactions, and how to avoid problems with them.

Concurrent Transactions

Sometimes it is useful for an application to have multiple independent connections called *concurrent transactions*. Using concurrent transactions, an application can connect to several databases at the same time, and can establish several distinct connections to the same database.

The context APIs that are used for multiple-thread database access allow an application to use concurrent transactions. Each context created in an application is independent from the other contexts. This means you create a context, connect to a database using the context, and run SQL statements against the database without being affected by the activities such as running COMMIT or ROLLBACK statements of other contexts.

For example, suppose you are creating an application that allows a user to run SQL statements against one database, and keeps a log of the activities performed in a second database. Because the log must be kept up to date, it is necessary to issue a COMMIT statement after each update of the log, but you do not want the user's SQL statements affected by commits for the log. This is a perfect situation

for concurrent transactions. In your application, create two contexts: one connects to the user's database and is used for all the user's SQL; the other connects to the log database and is used for updating the log. With this design, when you commit a change to the log database, you do not affect the user's current unit of work.

Another benefit of concurrent transactions is that if the work on the cursors in one connection is rolled back, it has no affect on the cursors in other connections. After the rollback in the one connection, both the work done and the cursor positions are still maintained in the other connections.

Related concepts:

• "Purpose of Multiple-Thread Database Access" on page 169

Potential Problems with Concurrent Transactions

An application that uses concurrent transactions can encounter some problems that cannot arise when writing an application that uses a single connection. When writing an application with concurrent transactions, exercise caution with the following:

- Database dependencies between two or more contexts.

  Each context in an application has its own set of database resources, including locks on database objects. These different sets of resources make it possible for two contexts, if they are accessing the same database object, to become deadlocked. The database manager will detect the deadlock, one of the contexts will receive an SQLCODE -911, and its unit of work will be rolled back.
- Application dependencies between two or more contexts.

  Switching contexts within a single thread creates dependencies between the contexts. If the contexts also have database dependencies, it is possible for a deadlock to develop. Because some of the dependencies are outside of the database manager, the deadlock will not be detected and the application will be suspended.

  As an example of this sort of problem, consider the following application:

```
context 1
UPDATE TAB1 SET COL = :new_val

context 2
SELECT * FROM TAB1
COMMIT

context 1
COMMIT
```

  Suppose the first context successfully executes the UPDATE statement. The update establishes locks on all the rows of TAB1. Now context 2 tries to select all the rows from TAB1. Because the two contexts are independent, context 2 waits on the locks held by context 1. Context 1, however, cannot release its locks until context 2 finishes executing. The application is now deadlocked, but the database manager does not know that context 1 is waiting on context 2, so it will not force one of the contexts to be rolled back. The unresolved dependency leaves the application suspended.

Related concepts:

• "Deadlock Prevention for Concurrent Transactions" on page 635

Deadlock Prevention for Concurrent Transactions

Because the database manager cannot detect deadlocks between contexts, you must design and code your application in a way that will prevent (or at least avoid) deadlocks. Consider the following example, which can result in a deadlock situation:

```
context 1
UPDATE TAB1 SET COL = :new_val

context 2
SELECT * FROM TAB1
COMMIT

context 1
COMMIT
```

Suppose the first context successfully executes the UPDATE statement. The update establishes locks on all the rows of TAB1. Now context 2 tries to select all the rows from TAB1. Because the two contexts are independent, context 2 waits on the locks held by context 1. Context 1, however, cannot release its locks until context 2 finishes executing. The application is now deadlocked, but the database manager does not know that context 1 is waiting on context 2, so it will not force one of the contexts to be rolled back. The unresolved dependency leaves the application suspended.

You can avoid the deadlock in the example in several ways:

- Release all locks held before switching contexts.

  Change the code so that context 1 performs its commit before switching to context 2.
- Do not access a given object from more than one context at a time.

  Change the code so that both the update and the select are done from the same context.
- Set the *locktimeout* database configuration parameter to a value other than -1.

  While a value other than -1 will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. When context 2 is rolled back, context 1 can continue executing (which releases the locks) and context 2 can retry its work.

Althought the techniques for avoiding deadlocks are described in terms of the example, you can apply them to all applications that use concurrent transactions.

Related concepts:

- "Potential Problems with Concurrent Transactions" on page 634

Related reference:

- "locktimeout - Lock timeout configuration parameter" in the *Administration Guide: Performance*

Savepoints and Transactions

The sections that follow describe savepoints, and how to use them to manage transactions.

Transaction management with savepoints

Application savepoints provide control over the work performed by a subset of SQL statements in a transaction or unit of work. Within your application you can set a savepoint, and later either release the savepoint or roll back the work performed since you set the savepoint. You can use as many savepoints as you require within a single transaction. The following example demonstrates the use of two savepoints within a single transaction to control the behavior of an application:

Example of an order using application savepoints::

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

-- set the first savepoint in the transaction
SAVEPOINT before_radio ON ROLLBACK RETAIN® CURSORS
  INSERT INTO order_item ... Radio
  INSERT INTO order_item ... Power Cord
  -- Pseudo-SQL:
  IF SQLSTATE = "No Power Cord"
    ROLLBACK TO SAVEPOINT before_radio
RELEASE SAVEPOINT before_radio

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

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

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

When you issue a ROLLBACK TO SAVEPOINT statement, the corresponding savepoint is not automatically released. Any subsequent SQL statements are associated with that savepoint, until the savepoint is released either explicitly with a RELEASE SAVEPOINT statement or implicitly by ending the transaction or unit of work. The savepoint can also be implicitly released at the end of the current savepoint level, or until a prior active savepoint is released or rolled back, at which point the current savepoint would become obsolete. This means that you can issue multiple ROLLBACK TO SAVEPOINT statements for a single savepoint.

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

Example of an order using multiple transactions::

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

INSERT INTO order_item ... Radio
INSERT INTO order_item ... Power Cord
-- Pseudo-SQL:
IF SQLSTATE = "No Power Cord"
  -- roll back current transaction, start new transaction
  ROLLBACK
ELSE
  -- commit current transaction, start new transaction
  COMMIT

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

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

Related samples:

- "tbsavept.sqc -- How to use external savepoints (C)"

Application Savepoints Compared to Compound SQL Blocks

Savepoints offer the following advantages over compound SQL blocks:

- Enhanced control of transactions
- Less locking contention
- Improved integration with application logic

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

Example of an order using application savepoints::

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

```
-- set the first savepoint in the transaction
SAVEPOINT before_radio ON ROLLBACK RETAIN® CURSORS
  INSERT INTO order_item ... Radio
  INSERT INTO order_item ... Power Cord
  -- Pseudo-SQL:
  IF SQLSTATE = "No Power Cord"
    ROLLBACK TO SAVEPOINT before_radio
RELEASE SAVEPOINT before_radio

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

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

When you issue a compound SQL block, DB2® simultaneously acquires the locks needed for the entire compound SQL block of statements. When you set an application savepoint, DB2 acquires locks as each statement in the scope of the savepoint is issued. The locking behavior of savepoints can lead to significantly less locking contention than compound SQL blocks, so unless your application requires the locking performed by compound SQL statements, it may be best to use savepoints.

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

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

Example of integrating savepoints and SQL statements within application logic:

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

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

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

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

    -- Pseudo-SQL:
    IF SQLSTATE = "No Power Cord"
      ROLLBACK TO SAVEPOINT before_radio
  COMMIT
}
```

SQL Statements for creating and controlling savepoints

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

SAVEPOINT
To set a savepoint, issue a SAVEPOINT SQL statement. To identify a specific savepoint for nested savepoints and to improve the clarity of your code, you can choose a meaningful name for the savepoint. These names are known as savepoint references. For example:

```
SAVEPOINT before_sales ON ROLLBACK RETAIN CURSORS
```

RELEASE SAVEPOINT
To release a savepoint, issue a RELEASE SAVEPOINT SQL statement. For example:

```
RELEASE SAVEPOINT before_sales
```

If you do not explicitly release a savepoint with a RELEASE SAVEPOINT SQL statement, it is released at the end of the current savepoint level.

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

```
ROLLBACK TO SAVEPOINT before_sales
```

A savepoint level refers to the scope of reference for any savepoint-related statement. When a savepoint level is started, no savepoint-related statement can refer to a savepoint created outside the new savepoint level. Similarly, savepoint references are resolved within the current savepoint level and do not take into account savepoint references outside of the current savepoint level.

A new savepoint level is started or entered only when any the following happens:

- A new unit of work is started
- A stored procedure defined with the NEW SAVEPOINT LEVEL clause
- An atomic compound SQL statement is started

A savepoint level is ended when the event that caused its creation is finished or removed.

The following rules apply to actions within a savepoint level's scope:

- Savepoints can only be referenced within the savepoint level in which they are established. You cannot release or rollback to a savepoint established outside of the current savepoint level.
- All active savepoints established within the current savepoint level are automatically released when the savepoint level ends.

- Savepoint unique names are only enforced within the current savepoint level. The names of savepoints that are active in surrounding savepoint levels can be reused in the current savepoint level without affecting these other savepoints.

Related reference:

- "ROLLBACK statement" in the *SQL Reference, Volume 2*
- "RELEASE SAVEPOINT statement" in the *SQL Reference, Volume 2*
- "SAVEPOINT statement" in the *SQL Reference, Volume 2*

Related samples:

- "tbsavept.sqc -- How to use external savepoints (C)"

Restrictions on Savepoint Usage

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

Triggers
DB2 does not support savepoint-related SQL statements within the definition body of a trigger. However, you can use a trigger to call stored procedures that contain savepoints. These stored procedures must be defined as starting a new savepoint level when invoked (this is done with the NEW SAVEPOINT LEVEL clause of the CREATE PROCEDURE statement).

SET INTEGRITY statement
Within a savepoint, DB2 treats SET INTEGRITY statements as DDL statements.

Related concepts:

- "Savepoints and Data Definition Language (DDL)" on page 640

Savepoints and Data Definition Language (DDL)

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

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

```
SAVEPOINT savepoint_name;
   PREPARE s1 FROM 'SELECT FROM t1';
   --issue DDL statement for t1
      ALTER TABLE t1 ADD COLUMN...
   PREPARE s2 FROM 'SELECT FROM t2';
   --issue DDL statement for t3
      ALTER TABLE t3 ADD COLUMN...
   PREPARE s3 FROM 'SELECT FROM t3';
   OPEN c1 USING s1;
   OPEN c2 USING s2;
   OPEN c3 USING s3;
```

```
ROLLBACK TO SAVEPOINT
FETCH c1; --invalid (SQLCODE -910)
FETCH c2; --successful
FETCH c3; --invalid (SQLCODE -910)
```

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

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

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

NOT LOGGED INITIALLY tables
If you create a table with the NOT LOGGED INITIALLY property, or alter a table to have the NOT LOGGED INITIALLY property within a savepoint, DB2 treats a ROLLBACK TO SAVEPOINT statement as a ROLLBACK WORK statement, and rolls back the entire unit of work.

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

DECLARE TEMPORARY TABLE outside savepoint
If a temporary table is declared outside a savepoint, a ROLLBACK TO SAVEPOINT statement does not drop the temporary table. If the temporary table is declared as logged (if the NOT LOGGED clause is not used in the DECLARE GLOBAL TEMPORARY TABLE statement), the changes made to the table within the savepoint will be rolled back. If the temporary table is declared as NOT LOGGED and the data in the table has been changed within the savepoint, all the rows will be deleted. Otherwise, if the data has not been changed, all the rows will be preserved.

Nesting savepoints

DB2® supports nested savepoints, where you can set up a savepoint within another savepoint. You can set an unlimited number of savepoints and there is also no limit to the number of nested savepoint levels.

This example demonstrates the use of nested savepoints:

```
CREATE TABLE Department (deptno CHAR(6), deptname VARCHAR(20), mgrno INT)
INSERT INTO Department VALUES ('A20', 'Marketing', 301)
SAVEPOINT savepoint1 ON ROLLBACK RETAIN® CURSORS
INSERT INTO Department VALUES ('B30', 'Finance', 520)
SAVEPOINT savepoint2 ON ROLLBACK RETAIN CURSORS
INSERT INTO Department VALUES ('C40', 'IT Support', 430)
SAVEPOINT savepoint3 ON ROLLBACK RETAIN CURSORS
INSERT INTO Department VALUES ('R50', 'Research', 150)
ROLLBACK TO SAVEPOINT savepoint3
ROLLBACK TO SAVEPOINT savepoint1
```

There are two levels of nesting in this example, with `savepoint3` nested in `savepoint2`, and `savepoint2` nested in `savepoint1`. In `savepoint3`, once the insertion of department `'R50'` takes place, a total of four rows will have been

added to the Department table in this example. When the ROLLBACK to savepoint3 statement is issued, only the insertion in savepoint3 is rolled back, leaving three rows in the Department table. When the ROLLBACK to savepoint1 statement is issued, all the work in savepoint2, and savepoint1 is rolled back, leaving one row in the Department table.

Savepoints and Buffered Inserts

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

Related concepts:

- "Buffered Inserts in Partitioned Database Environments" on page 655

Related reference:

- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

Savepoints and Cursor Blocking

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

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

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

When your application issues the first FETCH on table "t1", the DB2® server sends a block of column values (1, 2, and 3) to the client application. These column values are stored locally by the client. When your application issues the ROLLBACK TO SAVEPOINT SQL statement, column values '2' and '3' are deleted from the table. After the ROLLBACK TO SAVEPOINT statement, the next FETCH from the table returns column value '2', even though that value no longer exists in the table. The application receives this value because it takes advantage of the cursor blocking option to improve performance and accesses the data that it has stored locally.

Related reference:

- "BIND Command" in the *Command Reference*
- "PRECOMPILE Command" in the *Command Reference*

Savepoints and XA-Compliant Transaction Managers

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

X/Open XA Interface Programming Considerations

The X/Open XA Interface is an open standard for coordinating changes to multiple resources, while ensuring the integrity of these changes. Software products known as *transaction processing monitors* typically use the XA interface, and because DB2 supports this interface, one or more DB2 databases may be concurrently accessed as resources in such an environment.

Special consideration is required by DB2 when operating in a distributed transaction processing (DTP) environment that uses the XA interface, because a different model is used for transaction processing as compared to applications running independently of a TP monitor. The characteristics of this transaction processing model are:

- Multiple types of recoverable resources (such as DB2 databases) can be modified within a transaction.
- Resources are updated using two-phase commit to ensure the integrity of the transactions being executed.
- Application programs send requests to commit or roll back a transaction to the TP monitor product rather than to the managers of the resources. For example, in a CICS® environment an application would issue `EXEC CICS SYNCPOINT` to commit a transaction, and issuing `EXEC SQL COMMIT` to DB2 would be invalid and unnecessary.
- Authorization to run transactions is screened by the TP monitor and related software, so resource managers such as DB2 treat the TP monitor as the single authorized user. For example, any use of a CICS transaction must be authenticated by CICS and the access privilege to the database must be granted to CICS rather than the end user who invokes the CICS application.
- Multiple programs (transactions) are typically queued and executed on a database server (which appears to DB2 to be a single, long-running application program).

Due to the unique nature of this environment, DB2 has special behavior and requirements for applications coded to run in it:

- Multiple databases can be connected to and updated within a unit of work, without consideration of distributed unit of work precompiler options or client settings.
- The DISCONNECT statement is disallowed, and will be rejected with SQLCODE -30090 (SQLSTATE 25000) if attempted.
- The RELEASE statement is not supported, and will be rejected with a -30090.
- COMMIT and ROLLBACK statements are not allowed within stored procedures accessed by a TP monitor transaction.

- When two-phase commit flows are explicitly disabled for a transaction (these are called *LOCAL* transactions in XA Interface terminology) only one database can be accessed within that transaction. This database cannot be a host, AS/400®, or iSeries database that is accessed using SNA connectivity. Local transactions to DB2® for OS/390® Version 5 using TCP/IP connectivity are supported.
- LOCAL transactions should issue SQL COMMIT or SQL ROLLBACK at the end of each transaction; otherwise, the transaction will be considered part of the next transaction that is processed.
- Switching between current database connections is done through the use of either SQL CONNECT or SQL SET CONNECTION. The authorization used for a connection cannot be changed by specifying a user ID or password on the CONNECT statement.
- If a database object such as a table, view, or index is not fully qualified in a dynamic SQL statement, it will be implicitly qualified with the single authentication ID that the TP monitor is executing under, rather than user's ID.
- Any use of DB2 COMMIT or ROLLBACK statements for transactions that are not LOCAL will be rejected. The following codes will be returned:
  - SQLCODE -925 (SQLSTATE 2D521) for static COMMIT
  - SQLCODE -926 (SQLSTATE 2D521) for static ROLLBACK
  - SQLCODE -426 (SQLSTATE 2D528) for dynamic COMMIT
  - SQLCODE -427 (SQLSTATE 2D529) for dynamic ROLLBACK
- CLI requests to COMMIT or ROLLBACK are also rejected.
- Handling database-initiated rollback:

  In a DTP environment, if an RM has initiated a rollback (for instance, due to a system error or deadlock) to terminate its own branch of a global transaction, it must not process any more requests from the same application process until a transaction manager-initiated sync point request occurs. This includes deadlocks that occur within a stored procedure. For the database manager, this means rejecting all subsequent SQL requests with SQLCODE -918 (SQLSTATE 51021) to inform you that you must roll back the global transaction with the transaction manager's sync point service such as using the CICS SYNCPOINT ROLLBACK command in a CICS environment. If for some reason you request the TM to commit the transaction instead, the RM will inform the TM about the rollback and cause the TM to roll back other RMs anyway.
- Cursors declared WITH HOLD:

  Cursors declared WITH HOLD are supported in XA/DTP environments for CICS transaction processing monitors.

  In cases where cursors declared WITH HOLD are not supported, the OPEN statement will be rejected with SQLCODE -30090 (SQLSTATE 25000), reason code 03.

  It is the responsibility of the transactions to ensure that cursors specified to be WITH HOLD are explicitly closed when they are no longer required; otherwise, they might be inherited by other transactions, causing conflict or unnecessary use of resources.

  If the TP monitor supports WITH HOLD cursors, the `xa_commit`, `xa_rollback` and `xa_prepare` must be issued on the same connection as the global transaction.
- Statements that update or change a database are not allowed against databases that do not support two-phase commit request flows. For example, accessing host, AS/400, or iSeries database servers in environments in which level 2 of DRDA® protocol (DRDA2) is not supported.

- Whether a database supports updates in an XA environment can be determined at run-time by issuing a CONNECT statement. The third SQLERRD token will have the value 1 if the database is updatable; otherwise, this token will have the value 2.
- When updates are restricted, only the following SQL statements will be allowed:

```
CONNECT
DECLARE
DESCRIBE
EXECUTE IMMEDIATE (where the first token or keyword is SET but
                   not SET INTEGRITY)
OPEN CURSOR
FETCH CURSOR
CLOSE CURSOR
PREPARE (where the first token or keyword that is not blank or
         left parenthesis is SET (other than SET INTEGRITY),
         SELECT, WITH, or VALUES)
SELECT...INTO
VALUES...INTO
```

  Any other attempts will be rejected with SQLCODE -30090 (SQLSTATE 25000).

  The PREPARE statement will only be usable to prepare SELECT statements. The EXECUTE IMMEDIATE statement is also allowed to execute SQL SET statements that do not return any output value, such as the SET SQLID statement from DB2 Universal Database for z/OS and OS/390.
- API Restrictions:

  APIs that internally issue a commit in the database and bypass the two-phase commit process will be rejected with SQLCODE -30090 (SQLSTATE 25000). For a list of these APIs, see the article on restrictions on multisite update applications. These APIs are not supported in a multisite update (Connect Type 2).
- DB2 supports a multi-threaded XA/DTP environment.

Note that the above restrictions apply to applications running in a TP monitor environment that uses the XA interface. If DB2 databases are not defined for use with the XA interface, these restrictions do not apply; however, it is still necessary to ensure that transactions are coded in a way that will not leave DB2 in a state that will adversely affect the next transaction to be run.

Related concepts:

- "Security considerations for XA transaction managers" in the *Administration Guide: Planning*
- "Configuration considerations for XA transaction managers" in the *Administration Guide: Planning*
- "XA function supported by DB2 Universal Database" in the *Administration Guide: Planning*
- "Multisite Update with DB2 Connect" on page 700

Related tasks:

- "Updating host or iSeries database servers with an XA-compliant transaction manager" in the *Administration Guide: Planning*

Application Linkage and the X/Open XA Interface

To produce an executable application, you need to link in the application objects with the language libraries, the operating system libraries, the normal database manager libraries, and the libraries of the TP monitor and transaction manager products.

MTS and COM+ Transaction Management

Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager

DB2® UDB can be fully integrated with Microsoft® Transaction Server (MTS) Version 2.0 on Windows® NT or Microsoft Component Services (COM+) on Windows 2000 and Windows XP to coordinate two-phase commit with multiple DB2 UDB, zSeries®, and iSeries™ database servers, as well as with other resource managers that comply with MTS or COM+ specifications.

Prerequisites:

To use MTS or COM+ distributed transaction support, ensure that the following requirements are met for the Windows machine where the DB2 client is installed:

- Windows NT® with MTS at Version 2.0: Microsoft Hotfix 0772 or later

  MTS Version 2.0 for Windows NT is available as part of the Windows NT 4.0 Option Pack. You can download the Option Pack from:

  ```
  http://www.microsoft.com/ntserver/nts/downloads/recommended/NT4OptPk/
  ```

- Windows 2000: Service Pack 3 or later

For DB2 CLI applications using MTS or COM+:

- Do not change the default value of the SQL_ATTR_CONNECTION_POOLING CLI environment attribute (default SQL_CP_OFF)
- The installation of the DB2 ODBC driver on Windows operating systems will automatically add a new keyword to the registry:

  ```
  HKEY_LOCAL_MACHINE\software\ODBC\odbcinit.ini\IBM DB2 ODBC Driver:
  Keyword Value Name: CPTimeout
  Data Type: REG_SZ
  Value: 60
  ```

Supported DB2 database servers:

The following servers are supported for multisite update using MTS or COM+ coordinated transactions:

- DB2 Universal Database™ Enterprise Server Edition (ESE)

  Note: Loosely coupled global transactions for MTS or COM+ are not supported in massively parallel processing (MPP) environments. Loosely coupled global transactions exist when each of a number of application processes accesses resource managers as if it was in a separate global transaction, however, those application processes are under the coordination of the transaction manager. Each application process will have its own transaction branch within a resource manager. When a commit or rollback is requested by any one of the application processes, transaction manager,

or resource manager, the transaction branches are completed altogether. It is the application's responsibility to ensure that resource deadlock does not occur among the branches.

(Tightly coupled global transactions exist when multiple application processes take turns to do work under the same transaction branch in a resource manager. To the resource manager, the two application processes are a single entity. The resource manager must ensure that resource deadlock does not occur within the transaction branch.)

- DB2 Universal Database for z/OS™
- DB2 Universal Database for iSeries
- DB2 Server for VSE & VM

Installation and configuration considerations:

The following is a summary of installation and configuration considerations for using MTS (COM+ should be installed by default as part of Windows 2000):

- Install MTS and the DB2 client on the same machine where the MTS application runs.
- If host or iSeries database servers are involved in a multisite update:
  1. Install DB2 Connect™ functionality (either DB2 Connect Enterprise Edition (EE) or DB2 UDB Enterprise Server Edition (ESE) with the DB2 Connect functionality installed) on your local machine or on a remote machine. DB2 Connect functionality allows host or iSeries database servers to participate in a multisite update transaction.
  2. Ensure that your DB2 Connect server is enabled for multisite update.

Related concepts:

- "X/Open distributed transaction processing model" in the *Administration Guide: Planning*
- "MTS and COM+ Distributed Transaction Support and the IBM OLE DB Provider" on page 239
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) transaction timeout" on page 648
- "Loosely coupled support with Microsoft Component Services (COM+)" on page 648
- "ODBC and ADO connection pooling with Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+)" on page 649

Related tasks:

- "Installing DB2 Connect Enterprise Edition (Windows)" in the *Quick Beginnings for DB2 Connect Enterprise Edition*

Related reference:

- "SQLSetConnectAttr function (CLI) - Set connection attributes" in the *CLI Guide and Reference, Volume 2*
- "DB2 Connect product offerings" in the *DB2 Connect User's Guide*
- "Connection attributes (CLI) list" in the *CLI Guide and Reference, Volume 2*
- "Environment attributes (CLI) list" in the *CLI Guide and Reference, Volume 2*

Loosely coupled support with Microsoft Component Services (COM+)

Loosely coupled global transactions exist when each of a number of application processes accesses resource managers as if it was in a separate global transaction, however, those application processes are under the coordination of the transaction manager. Each application process will have its own transaction branch within a resource manager. When a commit or rollback is requested by any one of the application processes, transaction manager, or resource manager, the transaction branches are completed altogether. It is the application's responsibility to ensure that resource deadlock does not occur among the branches.

DB2® Universal Database Version 8 supports loosely coupled global transactions for COM+ objects, with no lock timeout or deadlock, given the following restrictions:

- Data definition language (DDL) is supported if it is executed on a single branch while no other loosely coupled transactions are active. If a loosely coupled branch attempts to start while a single branch executing DDL is active, the loosely coupled branch will be rejected. Conversely, if there is at least one active loosely coupled transaction, then any attempts to execute DDL on another branch will be rejected.
- Loosely coupled global transactions are not supported on massively parallel processing (MPP) environments. In an MPP environment, each global transaction is treated in isolation, where deadlock or timeout might occur.
- Savepoint processing and SQL statements are executed serially across multiple connections.
- When an implicit rollback has been performed on one connection, all branches on other connections that are related to the loosely coupled transaction will return SQL0998N, with reason code: 225 and subcode 4: ″Only rollbacks are allowed for this transaction″.

Related concepts:

- "X/Open distributed transaction processing model" in the *Administration Guide: Planning*
- "MTS and COM+ Distributed Transaction Support and the IBM OLE DB Provider" on page 239
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager" on page 646
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) transaction timeout" on page 648
- "ODBC and ADO connection pooling with Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+)" on page 649

Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) transaction timeout

Transaction timeout can be set through the following tools when MTS or COM+ is used:

- MTS (Microsoft Windows® NT): MTS Explorer tool
- COM+ (Microsoft Windows 2000 and XP): Component Services, located under Administrative Tools of the Windows Control Panel

If a transaction takes longer than the transaction timeout value (the default value is 60 seconds), MTS or COM+ will asynchronously issue an abort to all Resource Managers involved, and the entire transaction is aborted.

The abort is translated into a DB2® rollback request at the server. The rollback request is serialized on the connection, on servers other than DB2 for z/OS™ and DB2 for iSeries™, to guarantee the integrity of the data on the database server. When the server is DB2 for z/OS or DB2 for iSeries, then the connection should be defined with the INTERRUPT_ENABLED option in the DCS catalog entry so that when a timeout occurs, the connection from the DB2 Connect™ server to the z/OS or iSeries server will be disconnected, forcing a rollback on the z/OS or iSeries server.

As a result:

- If the connection is idle, the rollback is executed immediately.
- If a long-running SQL statement is processing, the rollback request waits until the SQL statement finishes.

Related concepts:

- "X/Open distributed transaction processing model" in the *Administration Guide: Planning*
- "DCS directory values" in the *DB2 Connect User's Guide*
- "MTS and COM+ Distributed Transaction Support and the IBM OLE DB Provider" on page 239
- "Processing of Interrupt Requests" on page 694
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager" on page 646
- "Loosely coupled support with Microsoft Component Services (COM+)" on page 648
- "ODBC and ADO connection pooling with Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+)" on page 649

ODBC and ADO connection pooling with Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+)

Connection pooling enables an application to use a connection from a pool of connections, so that the connection does not need to be re-established for each use. Once a connection has been created and placed in a pool, an application can reuse that connection without performing a complete connection process. The connection is pooled when the application disconnects from the data source and will be given to a new connection whose attributes are the same.

ODBC connection pooling:

Connection pooling has been a feature of the ODBC Driver Manager since ODBC 2.x. With the latest ODBC Driver Manager (version 3.5) available as part of the Microsoft® Data Access Components (MDAC) download, connection pooling has some configuration changes and new behavior for ODBC connections of transactional MTS COM+ objects.

The ODBC Driver Manager 3.5 requires that the ODBC driver register a new keyword in the registry before it allows connection pooling to be activated. The keyword is:

```
Key Name: SOFTWARE\ODBC\ODBCINST.INI\IBM DB2® ODBC DRIVER
Name: CPTimeout
Type: REG_SZ
Data: 60
```

The DB2 ODBC driver for the Windows® operating system fully supports connection pooling; therefore, this keyword is registered.

The default value of 60 means that the connection will be pooled for 60 seconds before it is disconnected.

In a busy environment, it is better to increase the CPTimeout value to a large number to prevent too many physical connects and disconnects, because these consume large amounts of system resource, including system memory and communications stack resources.

In addition, to ensure that the same connection is used between objects in the same transaction in a multiple processor machine, you must turn off "multiple pool per processor" support. To do this, copy the following registry setting into a file called `odbcpool.reg`, save it as a plain text file, and issue the command **odbcpool.reg**. The Windows operating system will import this registry setting.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Connection Pooling]
"NumberOfPools"="1"
```

Without this keyword set to 1, MTS or COM+ may pool connections for the same transaction in different pools, and hence may not reuse the same connection.

ADO connection pooling:

If the MTS or COM+ objects use ADO to access the database, you must turn off the OLE DB resource pooling so that the Microsoft OLE DB provider for ODBC (MSDASQL) will not interfere with ODBC connection pooling. This feature was initialized to `OFF` in ADO 2.0, but is initialized to `ON` in ADO 2.1. To turn OLE DB resource pooling off, copy the following lines into a file called `oledb.reg`, save it as a plain text file, and issue the command **oledb.reg**. The Windows operating system will import these registry settings.

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{c8b522cb-5cf3-11ce-ade5-00aa0044773d}]
@="MSDASQL"
"OLEDB_SERVICES"=dword:fffffffc
```

Related concepts:

- "X/Open distributed transaction processing model" in the *Administration Guide: Planning*
- "MTS and COM+ Distributed Transaction Support and the IBM OLE DB Provider" on page 239
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) as transaction manager" on page 646
- "Microsoft Transaction Server (MTS) and Microsoft Component Services (COM+) transaction timeout" on page 648

- "Loosely coupled support with Microsoft Component Services (COM+)" on page 648

Chapter 31. Programming Considerations for Partitioned Database Environments

FOR READ ONLY Cursors in a Partitioned Database Environment

If you declare a cursor from which you intend only to read, include FOR READ ONLY or FOR FETCH ONLY in the OPEN CURSOR declaration. (FOR READ ONLY and FOR FETCH ONLY are equivalent statements.) FOR READ ONLY cursors allow the coordinator partition to retrieve multiple rows at a time, dramatically improving the performance of subsequent FETCH statements. When you do not explicitly declare cursors FOR READ ONLY, the coordinator partition treats them as updatable cursors. Updatable cursors incur considerable expense because they require the coordinator partition to retrieve only a single row per FETCH.

Directed DSS and Local Bypass

The sections that follow describe considerations for using directed DSS and local bypass in partitioned database environments.

Directed DSS and Local Bypass in Partitioned Database Environments

To optimize online transaction processing (OLTP) applications, you may want to avoid simple SQL statements that require processing on all database partitions. You should design the application so that SQL statements can retrieve data from single database partitions. The directed distributed subsection (DSS) and local bypass techniques avoid the expense the coordinator partition incurs communicating with one or all of the associated partitions.

Related concepts:

- "Directed DSS in Partitioned Database Environments" on page 653
- "Local Bypass in Partitioned Database Environments" on page 654

Directed DSS in Partitioned Database Environments

A distributed subsection (DSS) is the action of sending subsections to the database partition that needs to do some work for a parallel query. It also describes the

initiation of subsections with invocation-specific values, such as values of variables in an OLTP environment. A *directed DSS* uses the table partitioning key to direct a query to a single partition. Use this type of query in your application to avoid the coordinator partition overhead required for a query broadcast to all partitions.

An example SELECT statement fragment that can take advantage of directed DSS follows:

```
SELECT ... FROM t1
  WHERE PARTKEY=:hostvar
```

When the coordinator partition receives the query, it determines which database partition holds the subset of data for *:hostvar*, and directs the query specifically to that database partition.

To optimize your application using directed DSS, divide complex queries into multiple simple queries. For example, in the following query the coordinator partition matches the partitioning key with multiple values. Because the data that satisfies the query lies on multiple database partitions, the coordinator partition broadcasts the query to all database partitions:

```
SELECT ... FROM t1
  WHERE PARTKEY IN (:hostvar1, :hostvar2)
```

Instead, break the query into multiple SELECT statements (each with a single host variable), or use a single SELECT statement with a UNION to achieve the same result. The coordinator partition can take advantage of simpler SELECT statements to use directed DSS to communicate only to the necessary database partitions. The optimized query looks like:

```
SELECT ... AS res1 FROM t1
  WHERE PARTKEY=:hostvar1
  UNION
SELECT ... AS res2 FROM t1
  WHERE PARTKEY=:hostvar2
```

Note that the above technique will only improve performance if the number of selects in the UNION is significantly smaller than the number of partitions.

Local Bypass in Partitioned Database Environments

A specialized form of the directed DSS query accesses data stored only on the coordinator partition. This is called a *local bypass* because the coordinator partition completes the query without having to communicate with another partition.

Local bypass is enabled automatically whenever possible, but you can increase its use by routing transactions to the database partition containing the data for that transaction. One technique for doing this is to have a remote client maintain connections to each database partition. A transaction can then use the correct connection based on the input partitioning key. Another technique is to group transactions by database partition and have a separate application server for each database partition.

To determine the number of the database partition on which the transaction data resides, you can use the `sqlugrpn` API (Get Row Partitioning Number). This API allows an application to efficiently calculate the partition number of a row, given the partitioning key.

Another alternative is to use the db2atld utility to divide input data by partition number and run a copy of the application against each database partition.

Related reference:

- "sqlugrpn - Get Row Partitioning Number" in the *Administrative API Reference*
- "db2atld - Autoloader Command" in the *Command Reference*

Buffered Inserts

The sections that follow describe considerations for using buffered inserts in partitioned database environments.

Buffered Inserts in Partitioned Database Environments

A buffered insert is an insert statement that takes advantage of table queues to buffer the rows being inserted, thereby gaining a significant performance improvement. To use a buffered insert, an application must be prepared or bound with the INSERT BUF option.

Buffered inserts can result in substantial performance improvement in applications that perform inserts. Typically, you can use a buffered insert in applications where a single insert statement (and no other database modification statement) is used within a loop to insert many rows and where the source of the data is a VALUES clause in the INSERT statement. Typically the INSERT statement is referencing one or more host variables that change their values during successive executions of the loop. The VALUES clause can specify a single row or multiple rows.

Typical decision support applications require the loading and periodic insertion of new data. This data could be hundreds of thousands of rows. You can prepare and bind applications to use buffered inserts when loading tables.

To cause an application to use buffered inserts, use the PREP command to process the application program source file, or use the BIND command on the resulting bind file. In both situations, you must specify the INSERT BUF option.

Note: Buffered inserts cause the following steps to occur:

1. The database manager opens one 4 KB buffer for each database partition on which the table resides.
2. The INSERT statement with the VALUES clause issued by the application causes the row (or rows) to be placed into the appropriate buffer (or buffers).
3. The database manager returns control to the application.
4. The rows in the buffer are sent to the partition when the buffer becomes full, or an event occurs that causes the rows in a partially filled buffer to be sent. A partially filled buffer is flushed when one of the following occurs:
   - The application issues a COMMIT (implicitly or explicitly through application termination) or ROLLBACK.
   - The application issues another statement that causes a savepoint to be taken. OPEN, FETCH, and CLOSE cursor statements do not cause a savepoint to be taken, nor do they close an open buffered insert.

     The following SQL statements will close an open buffered insert:
     – BEGIN COMPOUND SQL

- COMMIT
- DDL
- DELETE
- END COMPOUND SQL
- EXECUTE IMMEDIATE
- GRANT
- INSERT to a different table
- OPEN CURSOR for a full-select of a data change statement
- PREPARE of the same dynamic statement (by name) doing buffered inserts
- REDISTRIBUTE DATABASE PARTITION GROUP
- RELEASE SAVEPOINT
- REORG
- REVOKE
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- RUNSTATS
- SAVEPOINT
- SELECT INTO
- UPDATE
- Execution of any other statement, but not another (looping) execution of the buffered INSERT
- End of application

The following APIs will close an open buffered insert:

- BIND (API)
- REBIND (API)
- RUNSTATS (API)
- REORG (API)
- REDISTRIBUTE (API)

In any of these situations where another statement closes the buffered insert, the coordinator partition waits until every database partition receives the buffers and the rows are inserted. It then executes the other statement (the one closing the buffered insert), provided all the rows were successfully inserted.

The standard interface in a partitioned environment, (without a buffered insert) loads one row at a time doing the following steps (assuming that the application is running locally on one of the database partitions):

1. The coordinator partition passes the row to the database manager that is on the same partition.
2. The database manager uses indirect hashing to determine the database partition where the row should be placed:
   - The target partition receives the row.
   - The target partition inserts the row locally.
   - The target partition sends a response to the coordinator partition.
3. The coordinator partition receives the response from the target partition.
4. The coordinator partition gives the response to the application.

   The insertion is not committed until the application issues a COMMIT.
5. Any INSERT statement containing the VALUES clause is a candidate for buffered insert, regardless of the number of rows or the type of elements in the rows. That is, the elements can be constants, special registers, host variables, expressions, functions and so on.

For a given INSERT statement with the VALUES clause, the DB2® SQL compiler may not buffer the insert based on semantic, performance, or implementation considerations. If you prepare or bind your application with the INSERT BUF option, ensure that it is not dependent on a buffered insert. This means:

- Errors may be reported asynchronously for buffered inserts, or synchronously for regular inserts. If reported asynchronously, an insert error may be reported on a subsequent insert within the buffer, or on the *other* statement that closes the buffer. The statement that reports the error is not executed. For example, consider using a COMMIT statement to close a buffered insert loop. The commit reports an SQLCODE -803 (SQLSTATE 23505) due to a duplicate key from an earlier insert. In this scenario, the commit is not executed. If you want your application to really commit, for example, some updates that are performed before it enters the buffered insert loop, you must reissue the COMMIT statement.
- Rows inserted may be immediately visible through a SELECT statement using a cursor without a buffered insert. With a buffered insert, the rows will not be immediately visible. Do not write your application to depend on these cursor-selected rows if you precompile or bind it with the INSERT BUF option.

Buffered inserts result in the following performance advantages:

- Only one message is sent from the target partition to the coordinator partition for each buffer received by the target partition.
- A buffer can contain a large number of rows, especially if the rows are small.
- Parallel processing occurs as insertions are being done across partitions while the coordinator partition is receiving new rows.

An application that is bound with INSERT BUF should be written so that the same INSERT statement with VALUES clause is iterated repeatedly before any statement or API that closes a buffered insert is issued.

Note: You should do periodic commits to prevent the buffered inserts from filling the transaction log.

Related concepts:

- "Source File Creation and Preparation" on page 57
- "Package Creation Using the BIND Command" on page 64
- "Considerations for Using Buffered Inserts" on page 657
- "Restrictions on Using Buffered Inserts" on page 659

Considerations for Using Buffered Inserts

Buffered inserts exhibit behaviors that can affect an application program. This behavior is caused by the asynchronous nature of the buffered inserts. Based on the values of the row's partitioning key, each inserted row is placed in a buffer destined for the correct partition. These buffers are sent to their destination partitions as they become full, or an event causes them to be flushed. You must be aware of the following, and account for them when designing and coding the application:

- Certain error conditions for inserted rows are not reported when the INSERT statement is executed. They are reported later, when the first statement other than the INSERT (or INSERT to a different table) is executed, such as DELETE, UPDATE, COMMIT, or ROLLBACK. Any statement or API that closes the buffered insert statement can see the error report. Also, any invocation of the

insert itself may see an error of a previously inserted row. Moreover, if a buffered insert error is reported by another statement, such as UPDATE or COMMIT, DB2® will not attempt to execute that statement.

- An error detected during the insertion of a *group of rows* causes all the rows of that group to be backed out. A group of rows is defined as all the rows inserted through executions of a buffered insert statement:
  - From the beginning of the unit of work,
  - Since the statement was prepared (if it is dynamic), or
  - Since the previous execution of another updating statement. For a list of statements that close (or flush) a buffered insert, see the description of buffered inserts in partitioned database environments.
- An inserted row may not be immediately visible to SELECT statements issued after the INSERT by the same application program, if the SELECT is executed using a cursor.

A buffered INSERT statement is either open or closed. The first invocation of the statement opens the buffered INSERT, the row is added to the appropriate buffer, and control is returned to the application. Subsequent invocations add rows to the buffer, leaving the statement open. While the statement is open, buffers may be sent to their destination partitions, where the rows are inserted into the target table's partition. If any statement or API that closes a buffered insert is invoked while a buffered INSERT statement is open (including invocation of a *different* buffered INSERT statement), or if a PREPARE statement is issued against an open buffered INSERT statement, the open statement is closed before the new request is processed. If the buffered INSERT statement is closed, the remaining buffers are flushed. The rows are then sent to the target partitions and inserted. Only after all the buffers are sent and all the rows are inserted does the new request begin processing.

If errors are detected during the closing of the INSERT statement, the SQLCA for the new request will be filled in describing the error, and the new request is not done. Also, the entire group of rows that were inserted through the buffered INSERT statement *since it was opened* are removed from the database. The state of the application will be as defined for the particular error detected. For example:

- If the error is a deadlock, the transaction is rolled back (including any changes made before the buffered insert section was opened).
- If the error is a unique key violation, the state of the database is the same as before the statement was opened. The transaction remains active, and any changes made before the statement was opened are not affected.

For example, consider the following application that is bound with the buffered insert option:

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
     RESET COUNTER;
  END;
END;
EXEC SQL COMMIT;
```

Suppose the file contains 8 000 values, but value 3 258 is not legal (for example, a unique key violation). Each 1 000 inserts results in the execution of another SQL statement, which then closes the `INSERT INTO t2` statement. During the fourth

group of 1 000 inserts, the error for value 3 258 will be detected. It may be detected after the insertion of more values (not necessarily the next one). In this situation, an error code is returned for the `INSERT INTO t2` statement.

The error may also be detected when an insertion is attempted on table `t3`, which closes the `INSERT INTO t2` statement. In this situation, the error code is returned for the `INSERT INTO t3` statement, even though the error applies to table `t2`.

Suppose, instead, that you have 3 900 rows to insert. Before being told of the error on row number 3 258, the application may exit the loop and attempt to issue a COMMIT. The unique-key-violation return code will be issued for the COMMIT statement, and the COMMIT will not be performed. If the application wants to COMMIT the 3 000 rows that are in the database thus far (the last execution of `EXEC SQL INSERT INTO t3 ...` ends the savepoint for those 3 000 rows), the COMMIT has to be *reissued*. Similar considerations apply to ROLLBACK as well.

Note: When using buffered inserts, you should carefully monitor the SQLCODES returned to avoid having the table in an indeterminate state. For example, if you remove the `SQLCODE < 0` clause from the THEN DO statement in the above example, the table could end up containing an indeterminate number of rows.

Related concepts:

• "Buffered Inserts in Partitioned Database Environments" on page 655

Restrictions on Using Buffered Inserts

The following restrictions apply to buffered inserts:

- For an application to take advantage of the buffered inserts, one of the following must be true:
  - The application must either be prepared through PREP or bound with the BIND command and the INSERT BUF option is specified.
  - The application must be bound using the BIND or the PREP API with the SQL_INSERT_BUF option.
- If the INSERT statement with VALUES clause includes long fields or LOBS in the explicit or implicit column list, the INSERT BUF option is ignored for that statement and a normal insert section is done, not a buffered insert. This is not an error condition, and no error or warning message is issued.
- INSERT with fullselect is not affected by INSERT BUF. A buffered insert does not improve the performance of this type of INSERT.
- Buffered inserts can be used only in applications, and not through CLP-issued inserts, as these are done through the EXECUTE IMMEDIATE statement.

The application can then be run from any supported client platform.

Example of Extracting a Large Volume of Data in a Partitioned Database Environment

Although DB2 Universal Database provides excellent features for parallel query processing, the single point of connection of an application or an EXPORT command can become a bottleneck if you are extracting large volumes of data. This bottleneck occurs because the passing of data from the database manager to the application is a CPU-intensive process that executes on a single partition (typically a single processor as well).

DB2 Universal Database provides several methods to overcome the bottleneck, so that the volume of extracted data scales linearly per unit of time with an increasing number of processors. The following example describes the basic idea behind these methods.

Assume that you have a table called EMPLOYEE which is stored on 20 database partitions, and you generate a mailing list (FIRSTNME, LASTNAME, JOB) of all employees who are in a legitimate department (that is, WORKDEPT is not NULL).

The following query is run on each partition, then generates the entire answer set at a single partition (the coordinator partition):

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
```

But, the following query could be run on each partition for the database (that is, if there are five partitions, five separate queries are required, one at each partition). Each query generates the set of all the employee names whose record is on the particular partition where the query runs. Each local result set can be redirected to a file. The result sets then need to be merged into a single result set.

On AIX®, you can use a property of Network File System (NFS) files to automate the merge. If all the partitions direct their answer sets to the same file on an NFS mount, the results are merged. Note that using NFS without blocking the answer into large buffers results in very poor performance.

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
                AND NODENUMBER(NAME) = CURRENT NODE
```

The result can either be stored in a local file (meaning that the final result would be 20 files, each containing a portion of the complete answer set), or in a single NFS-mounted file.

The following example uses the second method, so that the result is in a single file that is NFS mounted across the 20 nodes. The NFS locking mechanism ensures serialization of writes into the result file from the different partitions. Note that this example, as presented, runs on the AIX platform with an NFS file system installed.

```
#define _POSIX_SOURCE
#define INCL_32

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sqlenv.h>
#include <errno.h>
#include <sys/access.h>
#include <sys/flock.h>
#include <unistd.h>
```

```
#define BUF_SIZE 1500000  /* Local buffer to store the fetched records */
#define MAX_RECORD_SIZE 80 /* >= size of one written record */

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

    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
       char dbname[10];  /* Database name (argument of the program) */
       char userid[9];
       char passwd[19];
       char first_name[21];
       char last_name[21];
       char job_code[11];
    EXEC SQL END DECLARE SECTION;

       struct flock unlock ;  /* structures and variables for handling */
       struct flock lock ;  /* the NFS locking mechanism */
       int lock_command ;
       int lock_rc ;
       int iFileHandle ;  /* output file */
       int iOpenOptions = 0 ;
       int iPermissions ;
       char * file_buf ;  /* pointer to the buffer where the fetched
                             records are accumulated */
       char * write_ptr ;  /* position where the next record is written */
       int buffer_len = 0 ;  /* length of used portion of the buffer */

    /* Initialization */

       lock.l_type = F_WRLCK;  /* An exclusive write lock request */
       lock.l_start = 0;  /* To lock the entire file */
       lock.l_whence = SEEK_SET;
       lock.l_len = 0;
       unlock.l_type = F_UNLCK;  /* An release lock request */
       unlock.l_start = 0;  /* To unlock the entire file */
       unlock.l_whence = SEEK_SET;
       unlock.l_len = 0;
       lock_command = F_SETLKW;  /* Set the lock */
       iOpenOptions = O_CREAT;  /* Create the file if not exist */
       iOpenOptions |= O_WRONLY;  /* Open for writing only */

    /* Connect to the database */

       if (argc == 3) {
          strcpy( dbname, argv[2] ); /* get database name from the argument */
          EXEC SQL CONNECT TO :dbname IN SHARE MODE ;
          if ( SQLCODE != 0 ) {
             printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
               SQLCODE );
       exit(1);
   }
       }
       else if ( argc == 5 ) {
         strcpy( dbname, argv[2] ); /* get  database name from the argument */
  strcpy (userid, argv[3]);
  strcpy (passwd, argv[4]);
  EXEC SQL CONNECT TO :dbname IN SHARE MODE USER :userid USING :passwd;
  if ( SQLCODE != 0 ) {
     printf( "Error: CONNECT TO the database failed. SQLCODE = %ld\n",
             SQLCODE );
       exit( 1 );
         }
       }
       else {
    printf ("\nUSAGE: largevol txt_file database [userid passwd]\n\n");
    exit( 1 ) ;
       } /* endif */
```

```
  /* Open the input file with the specified access permissions */

 if ( ( iFileHandle = open(argv[1], iOpenOptions, 0666 ) ) == -1 ) {
   printf( "Error: Could not open %s.\n", argv[2] ) ;
   exit( 2 ) ;
}

/* Set up error and end of table escapes */

EXEC SQL WHENEVER SQLERROR GO TO ext ;
EXEC SQL WHENEVER NOT FOUND GO TO cls ;

/* Declare and open the cursor */

EXEC SQL DECLARE c1 CURSOR FOR
         SELECT firstnme, lastname, job FROM employee
         WHERE workdept IS NOT NULL
         AND NODENUMBER(lastname) = CURRENT NODE;
EXEC SQL OPEN c1 ;

/* Set up the temporary buffer for storing the fetched result */

if ( ( file_buf = ( char * ) malloc( BUF_SIZE ) ) == NULL ) {
   printf( "Error: Allocation of buffer failed.\n" ) ;
   exit( 3 ) ;
}
memset( file_buf, 0, BUF_SIZE ) ; /* reset the buffer */
buffer_len = 0 ;  /* reset the buffer length */
write_ptr = file_buf ;  /* reset the write pointer */
/* For each fetched record perform the following    */
/*  - insert it into the buffer following the       */
/*    previously stored record                      */
/*  - check if there is still enough space in  the  */
/*    buffer for the next record and lock/write/    */
/*    unlock the file and initialize the buffer     */
/*    if not                                        */

do {
   EXEC SQL FETCH c1 INTO :first_name, :last_name, :job_code;
    buffer_len += sprintf( write_ptr, "%s %s %s\n",
                           first_name, last_name, job_code );
    buffer_len = strlen( file_buf ) ;
   /* Write the content of the buffer to the file if */
   /* the buffer reaches the limit                   */
   if ( buffer_len >= ( BUF_SIZE - MAX_RECORD_SIZE ) ) {
    /*  get excl. write lock */
    lock_rc = fcntl( iFileHandle, lock_command, &lock );
 if ( lock_rc != 0 ) goto file_lock_err;
 /*  position at the end of file */
 lock_rc = lseek( iFileHandle, 0, SEEK_END );
if ( lock_rc < 0 ) goto file_seek_err;
/* write the buffer */
lock_rc = write( iFileHandle,
                           ( void * ) file_buf, buffer_len );
if ( lock_rc < 0 ) goto file_write_err;
  /* release the lock */
 lock_rc = fcntl( iFileHandle, lock_command, &unlock );
 if ( lock_rc != 0 ) goto file_unlock_err;
 file_buf[0] = '\0' ;  /* reset the buffer */
  buffer_len = 0 ;  /* reset the buffer length */
  write_ptr = file_buf ;  /* reset the write pointer */
   }
   else {
      write_ptr = file_buf + buffer_len ;  /* next write position */
   }
  } while (1) ;
```

```
cls:
    /* Write the last piece of data out to the file */
    if (buffer_len > 0) {
       lock_rc = fcntl(iFileHandle, lock_command, &lock);
       if (lock_rc != 0) goto file_lock_err;
      lock_rc = lseek(iFileHandle, 0, SEEK_END);
       if (lock_rc < 0) goto file_seek_err;
       lock_rc = write(iFileHandle, (void *)file_buf, buffer_len);
       if (lock_rc < 0) goto file_write_err;
       lock_rc = fcntl(iFileHandle, lock_command, &unlock);
      if (lock_rc != 0) goto file_unlock_err;
    }
    free(file_buf);
 close(iFileHandle);
    EXEC SQL CLOSE c1;
    exit (0);
 ext:
    if ( SQLCODE != 0 )
       printf( "Error:  SQLCODE = %ld.\n", SQLCODE );
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CONNECT RESET;
    if ( SQLCODE != 0 ) {
       printf( "CONNECT RESET Error:  SQLCODE = %ld\n", SQLCODE );
       exit(4);
    }
    exit (5);
 file_lock_err:
    printf("Error: file lock error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(6);
file_seek_err:
    printf("Error: file seek error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
   exit(7);
file_write_err:
    printf("Error: file write error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(8);
file_unlock_err:
    printf("Error: file unlock error = %ld.\n",lock_rc);
      /* unconditional unlock of the file */
   fcntl(iFileHandle, lock_command, &unlock);
    exit(9);
}
```

This method is applicable not only to a select from a single table, but also for more complex queries. If, however, the query requires noncollocated operations (that is, the Explain shows more than one subsection besides the Coordinator subsection), this can result in too many processes on some partitions if the query is run in parallel on all partitions. In this situation, you can store the query result in a temporary table TEMP on as many partitions as required, then do the final extract in parallel from TEMP.

If you want to extract all employees, but only for selected job classifications, you can define the TEMP table with the column names, FIRSTNME, LASTNAME, and JOB, as follows:

```
INSERT INTO TEMP
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
       AND EMPNO NOT IN (SELECT EMPNO FROM EMP_ACT WHERE
                         EMPNO<200)
```

Then you would perform the parallel extract on TEMP.

When defining the TEMP table, consider the following:

- If the query specifies an aggregation GROUP BY, you should define the partitioning key of TEMP as a subset of the GROUP BY columns.
- The partitioning key of the TEMP table should have enough cardinality (that is, number of distinct values in the answer set) to ensure that the table is equally distributed across the partitions on which it is defined.
- Create the TEMP table with the NOT LOGGED INITIALLY attribute, then COMMIT the unit of work that created the table to release any acquired catalog locks.
- When you use the TEMP table, you should issue the following statements in a single unit of work:
  1. ALTER TABLE TEMP ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE (to empty the TEMP table and turn logging off)
  2. INSERT INTO TEMP SELECT FIRSTNAME...
  3. COMMIT

  This technique allows you to insert a large answer set into a table without logging and without catalog contention. However, if a table has the NOT LOGGED INITIALLY attribute activated, a non-logged activity occurs and either of the following situations occur::

  – A statement fails, causing a rollback

  – A ROLLBACK TO SAVEPOINT is executed

  the entire unit of work is rolled back (SQL1476N), resulting in an unusable TEMP table. If this occurs, you will have to drop and recreate the TEMP table. For this reason, you should *not* use this technique to add data to a table that you could not easily recreate.

If you require the final answer set (which is the merged partial answer set from all partitions) to be sorted, you can:

- Specify the SORT BY clause on the final SELECT
- Do an extract into a separate file on each partition
- Merge the separate files into one output set using, for example, the **sort -m** AIX command.

Creating a Simulated Partitioned Database Environment

You can create a test environment for your partitioned environment applications without setting up a partitioned database environment.

Procedure:

To create a simulated partitioned database environment:

1. Create a model of your database design with DB2 Enterprise Server Edition.
2. Create sample tables with the PARTITIONING KEY clause that you will use to distribute your data across partitions in the production environment.
3. Develop and run your applications against the test database.

DB2 Enterprise Server Edition enforces the partitioning key constraints that apply in a partitioned database environment, and provides a useful test environment for your applications.

Troubleshooting

The sections that follow describe how to troubleshoot applications in a partitioned database environment.

Error-Handling Considerations in Partitioned Database Environments

In a partitioned database environment, DB2® breaks up SQL statements into subsections, each of which is processed on the partition that contains the relevant data. As a result, an error may occur on a partition that does not have access to the application. This condition does not occur in a nonpartitioned database environment.

You should consider the following:
- Non-CURSOR (EXECUTE) non-severe errors
- CURSOR non-severe errors
- Severe errors
- Merged multiple SQLCA structures
- How to identify the partition that returned the error

If an application ends abnormally because of a severe error, indoubt transactions may be left in the database. (An indoubt transaction pertains to global transactions when one phase completes successfully, but the system fails before the subsequent phase can complete, leaving the database in an inconsistent state.)

Related concepts:
- "Severe Errors in Partitioned Database Environments" on page 665
- "Merged Multiple SQLCA Structures" on page 666
- "Partition That Returns the Error" on page 666

Related tasks:
- "Manually resolving indoubt transactions" in the *Administration Guide: Planning*

Severe Errors in Partitioned Database Environments

If a severe error occurs in a partitioned database environment, one of the following will occur:

- The database manager on the partition where the error occurs shuts down.

  Active units of work are not rolled back.

  In this situation, you must recover the partition and any databases that were active on the partition when the shutdown occurred.
- All agents are forced off the database at the partition where the error occurred.

  All units of work on that database are rolled back.

  In this situation, the database partition where the error occurred is marked as inconsistent. Any attempt to access it results in either SQLCODE -1034 (SQLSTATE 58031) or SQLCODE -1015 (SQLSTATE 55025) being returned. Before you or any other application on another partition can access this database partition, you must run the RESTART DATABASE command against the database.

The severe error SQLCODE -30081 (SQLSTATE 08001) can occur for a variety of reasons. If you receive this message, check the SQLCA, which will indicate which partition failed. Then check the administration notification log file for details.

Related concepts:

- "Partition That Returns the Error" on page 666

Related reference:

- "RESTART DATABASE Command" in the *Command Reference*

Merged Multiple SQLCA Structures

One SQL statement may be executed by a number of agents on different database partition, and each agent may return a different SQLCA for different errors or warnings. The coordinating agent also has its own SQLCA. In addition, the SQLCA also has fields that indicate global numbers (such as the *sqlerrd* fields that indicate row counts). To provide a consistent view for applications, all the SQLCA values are merged into one structure.

Error reporting is as follows:

- Severe error conditions are always reported. As soon as a severe error is reported, no additions beyond the severe error are added to the SQLCA.
- If no severe error occurs, a deadlock error takes precedence over other errors.
- For all other errors, the SQLCA for the first negative SQLCODE is returned to the application.
- If no negative SQLCODE values are detected, the SQLCA for the first warning (that is, positive SQLCODE) is returned to the application. The exception to this occurs if a data manipulation operation is issued on a table that is empty on one partition, but has data on other partitions. The SQLCODE +100 is only returned to the application if agents from all partitions return SQL0100W, either because the table is empty on all partitions or there are no rows that satisfy the WHERE clause in an UPDATE statement.
- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- The values in the *sqlerrd* fields that indicate row counts are accumulations from all agents.

An application may receive a subsequent error or warning after the problem that caused the first error or warning is corrected. Errors are reported to the SQLCA to ensure that the first error detected is given priority over others. This ensures that an error caused by an earlier error cannot overwrite the original error. Severe errors and deadlock errors are given higher priority because they require immediate action by the coordinating agent.

Related reference:

- "SQLCA" in the *Administrative API Reference*

Partition That Returns the Error

If a partition returns an error or warning, its number is in the SQLERRD(6) field of the SQLCA. The number in this field is the same as that specified for the partition in the `db2nodes.cfg` file.

If an SQL statement or API call is successful, the partition number in this field is not significant.

Related reference:

- "SQLCA" in the *Administrative API Reference*

Looping or Suspended Applications

It is possible that, after you start a query or application, you suspect that it is suspended (it does not show any activity) or that it is looping (it shows activity, but no results are returned to the application). Ensure that you have turned lock timeouts on. In some situations, however, no error is returned. In these situations, you may find the diagnostic tools provided with DB2® and the database system monitor snapshot helpful.

One of the functions of the database system monitor that is useful for debugging applications is to display the status of all active agents. To obtain the greatest use from a snapshot, ensure that statement collection is being done before you run the application (preferably immediately after you run DB2START) as follows:

```
db2_all "db2 UPDATE MONITOR SWITCHES USING STATEMENT ON"
```

When you suspect that your application or query is either stalled or looping, issue the following command:

```
db2_all "db2 GET SNAPSHOT FOR AGENTS ON database
```

Related concepts:

- "Database system monitor" in the *System Monitor Guide and Reference*
- "The database system monitor information" in the *Administration Guide: Performance*

Related reference:

- "GET SNAPSHOT Command" in the *Command Reference*
- "UPDATE MONITOR SWITCHES Command" in the *Command Reference*
- "db2trc - Trace Command" in the *Command Reference*
- "db2support - Problem Analysis and Environment Collection Tool Command" in the *Command Reference*

Chapter 32. Common DB2 Application Techniques

Running applications from the Windows Local System Account

On Windows platforms, DB2 supports running applications from the Local System Account (LSA) using an implicit local database connection. You cannot make an explicit database connection using the LSA, and you also cannot make a remote database connection using the LSA.

In DB2, the schema name for the LSA is SYSTEM. Since DB2 has restrictions on objects with a schema name that starts with "SYS", database applications that are run from the LSA cannot create DB2 objects using the implicit connection schema. To create DB2 objects from an application that will run from the LSA, you must do the following:

- For static SQL, bind the application with a QUALIFIER value other than "SYSTEM".
- For dynamic SQL, you should explicitly qualify the objects you are creating with a schema name other than "SYSTEM", or set the CURRENT SCHEMA register to a schema name other than "SYSTEM".

Related reference:

- "SET SCHEMA statement" in the *SQL Reference, Volume 2*
- "BIND Command" in the *Command Reference*
- "CURRENT SCHEMA special register" in the *SQL Reference, Volume 1*

Generated Columns

Rather than using cumbersome insert and update triggers, DB2® enables you to include generated columns in your tables using the GENERATED ALWAYS AS clause. A generated column is a column that derives the values for each row from an expression, rather than from an insert or update operation. While combining an update trigger and an insert trigger can achieve a similar effect, using a generated column guarantees that the derived value is consistent with the expression.

To create a generated column in a table, use the GENERATED ALWAYS AS clause for the column and include the expression from which the value for the column will be derived. You can include the GENERATED ALWAYS AS clause in ALTER

TABLE and CREATE TABLE statements. The following example creates a table with two regular columns, “c1” and “c2”, and two generated columns, “c3” and “c4”, that are derived from the regular columns of the table.

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

Related tasks:

- “Defining a generated column on a new table” in the *Administration Guide: Implementation*
- “Defining a generated column on an existing table” in the *Administration Guide: Implementation*

Related reference:

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

Related samples:

- “TbGenCol.java -- How to use generated columns (JDBC)”

Identity Columns

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

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

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

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

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

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

The IDENTITY clause is available on both the CREATE TABLE and ALTER TABLE statements.

Related concepts:

- "Identity columns" in the *Administration Guide: Planning*

Related tasks:

- "Defining an identity column on a new table" in the *Administration Guide: Implementation*
- "Modifying an identity column definition" in the *Administration Guide: Implementation*
- "Altering an identity column" in the *Administration Guide: Implementation*

Related reference:

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

Related samples:

- "tbident.sqc -- How to use identity columns (C)"
- "TbIdent.java -- How to use Identity Columns (JDBC)"
- "TbIdent.sqlj -- How to use Identity Columns (SQLj)"

Retrieval of result sets from an SQL data change statement

Applications that modify tables with INSERT, UPDATE, or DELETE statements might require additional processing on the modified rows. To facilitate this processing, you can embed SQL data-change operations in the FROM clause of SELECT and SELECT INTO statements. Within a single unit of work, applications can retrieve a result set containing the modified rows from a table or view modified by an SQL data-change operation.

For example, the following statement updates the salaries of all the records in the EMPLOYEE table in the SAMPLE database and then returns the employee number and new salary for all the updated rows.

```
SELECT empno, salary FROM FINAL TABLE
 (UPDATE employee SET salary = salary * 1.10 WHERE job = 'CLERK')
```

To return data successfully, SELECT statements that retrieve result sets FROM SQL data-change operations require the SQL data-change operations to run successfully. The success of SQL data-change operations includes the processing of all constraints and triggers, if applicable.

For instance, suppose a user with SELECT privileges, but without INSERT privileges on the EMPLOYEE table attempts a SELECT FROM INSERT statement

on the EMPLOYEE table. The INSERT operation fails because of the missing privileges, and as a result, the entire SELECT statement fails.

Consider the following query, where records from the EMPLOYEE table are selected and then inserted into a different table, named EMP. This SELECT statement will fail.

```
SELECT empno FROM FINAL TABLE
 (INSERT INTO emp(name, salary)
 SELECT firstnme || midinit || lastname, salary
 FROM employee)
```

If the EMPLOYEE table has 100 rows and row 90 has a SALARY value of $9,999,000.00, then the addition of $10,000.00 would cause a decimal overflow to occur. The overflow would force the database manager to rollback the insertions into the EMP table.

Intermediate result tables

The modified rows of the table or view targeted by an SQL data-change operation in the FROM clause of a SELECT statement compose an intermediate result table. The intermediate result table includes all the columns of the target table or view, in addition to any include columns defined in the SQL data-change operation. You can reference all of the columns in an intermediate result table by name in the select list, the ORDER BY clause, or the WHERE clause.

The contents of the intermediate result table are dependant on the qualifier specified in the FROM clause. You must include one of the following FROM clause qualifiers in SELECT statements that retrieve result sets as intermediate result tables.

OLD TABLE
The rows in the intermediate result table will contain values of the target table rows at the point immediately preceding the execution of before triggers and the SQL data-change operation. The OLD TABLE qualifier applies to UPDATE and DELETE operations.

NEW TABLE
The rows in the intermediate result table will contain values of the target table rows at the point immediately after the SQL data-change statement has been executed, but before referential integrity evaluation and the firing of any after triggers. The NEW TABLE qualifier applies to UPDATE and INSERT operations.

FINAL TABLE
This qualifier returns the same intermediate result table as NEW TABLE. In addition, the use of FINAL TABLE guarantees that no after trigger or referential integrity constraint will further modify the target of the UPDATE or INSERT operation. The FINAL TABLE qualifier applies to UPDATE and INSERT operations.

The FROM clause qualifiers determine what version of the targeted data is in the intermediate result table. These qualifiers do not affect the insertion, deletion, or updates of target table rows.

Target tables and views

When selecting result sets FROM SQL data-change operations, the target can be either a table or a view.

In SQL data-change operations against views, the result table cannot include rows that no longer satisfy the view definition for NEW TABLE and FINAL TABLE. If you embed an INSERT or UPDATE statement that references a view in a SELECT statement, the view must be defined as WITH CASCADED CHECK OPTION. Alternatively, the view must satisfy the restrictions that would allow you to define it as WITH CASCADED CHECK OPTION.

If the target of SQL data-change operations embedded in the FROM clause of a SELECT statement is a fullselect, the result table can include rows that no longer qualify in the fullselect. This is because the predicates in the WHERE clause are not re-evaluated against the updated values.

Result set sorting based on INPUT SEQUENCE

To SELECT rows in the same order as they are inserted into the target table or view, use the INPUT SEQUENCE keywords in the ORDER BY clause. Use of the INPUT SEQUENCE keywords does not force rows to be inserted in the same order they are provided.

The following example demonstrates the use of the INPUT SEQUENCE keywords in the ORDER BY clause to sort the result set of an INSERT operation.

```
CREATE TABLE orders (purchase_date DATE,
                     sales_person VARCHAR(16),
                     region VARCHAR(10),
                     quantity SMALLINT,
                     order_num INTEGER NOT NULL
                       GENERATED ALWAYS AS IDENTITY (START WITH 100,
                                     INCREMENT BY 1))

SELECT * FROM FINAL TABLE
         (INSERT INTO orders
          (purchase_date, sales_person, region, quantity)
          VALUES (CURRENT DATE,'Judith','Beijing',6),
                 (CURRENT DATE,'Marieke','Medway',5),
                 (CURRENT DATE,'Hanneke','Halifax',5))
         ORDER BY INPUT SEQUENCE

PURCHASE_DATE SALES_PERSON     REGION     QUANTITY ORDER_NUM
------------- ---------------- ---------- -------- -----------
07/18/2003    Judith           Beijing           6         100
07/18/2003    Marieke          Medway            5         101
07/18/2003    Hanneke          Halifax           5         102
```

You can also sort result set rows using include columns.

Related concepts:

- "Retrieval of result sets from SQL data change statements using cursors" on page 674
- "Include columns" on page 675

Related reference:

- "DELETE statement" in the *SQL Reference, Volume 2*
- "INSERT statement" in the *SQL Reference, Volume 2*
- "SELECT statement" in the *SQL Reference, Volume 2*
- "SELECT INTO statement" in the *SQL Reference, Volume 2*
- "UPDATE statement" in the *SQL Reference, Volume 2*

Retrieval of result sets from SQL data change statements using cursors

You can declare cursors for queries that retrieve result sets from SQL data-change operations. For example:

```
DECLARE C1 CURSOR FOR SELECT salary FROM FINAL TABLE
        (INSERT INTO employee (name, salary, level)
            SELECT name, income, band FROM old_employee)
```

Errors that occur when fetching from a cursor whose definition contains an SQL data-change operation will not cause a rollback of the modified rows. Even if the errors result in the cursor's closing, the row modifications will remain intact because they were completed when the application opened the cursor.

Upon the opening of such a cursor, the database manager completely executes the SQL data-change operation and the result set is stored in a temporary table. If an error occurs while the cursor opens, the changes made by the SQL data-change operation are rolled back. Further updates to the target table or view will not appear in the result table rows for cursors that retrieve result sets from SQL data-change operations. For example, an application declares a cursor, opens the cursor, performs a fetch, updates the table, and fetches additional rows. The fetches after the UPDATE statement will return those values that were determined during open cursor processing prior to the UPDATE statement.

You can declare scrollable cursors for queries that retrieve result sets from SQL data-change operations. Since the result table is generated when you OPEN the cursor, the data modifications have already been written to the target table or view. Cursors with queries that select rows from an SQL data change operation must be defined as INSENSITIVE or ASENSITIVE.

Note: Scrollable cursors are supported only in CLI, JDBC, and SQLJ applications.

If you declare a cursor with the WITH HOLD option and the application performs a COMMIT, all of the data changes are committed. Cursors that you do not declare as WITH HOLD behave in the same manner. For all cursors, the SQL data-change operation included in the query is completely evaluated before any row is fetched.

When performing an explicit rollback for an OPEN CURSOR statement, or when rolling back to a save point prior to an OPEN CURSOR statement, all of the data changes for that cursor will be undone. For cursors with queries that retrieve result sets from SQL data-change operations, all data changes are undone after a ollback, but the cursor is retained and the previously inserted rows can still be fetched.

Related concepts:

- "Retrieval of result sets from an SQL data change statement" on page 671
- "Include columns" on page 675

Related reference:

- "DECLARE CURSOR statement" in the *SQL Reference, Volume 2*
- "DELETE statement" in the *SQL Reference, Volume 2*
- "FETCH statement" in the *SQL Reference, Volume 2*
- "OPEN statement" in the *SQL Reference, Volume 2*
- "SELECT statement" in the *SQL Reference, Volume 2*
- "UPDATE statement" in the *SQL Reference, Volume 2*

Include columns

With include columns, you can introduce columns that do not exist in the target table or view into an intermediate result table. You can assign values to include columns to use as handles for rows in the intermediate result table. Include columns do not affect SQL data-change operations, nor do they change the definitions of target tables or views.

An include column can be of any data type, is nullable, and must have a name that is unique from any column in the target table or view in the SQL data-change operation. You can refer to include columns in the select list, ORDER BY clause, or WHERE clause of the SELECT statement. In result sets, include columns appear as the right-most columns.

Include columns in INSERT operations

To assign values to include columns in INSERT operations, you can use the VALUES clause. A common use for include columns in INSERT operations is to customize the ordering of result sets. For example:

```
SELECT * FROM FINAL TABLE
         (INSERT INTO sales INCLUDE (sortkey integer) VALUES
          (CURRENT DATE,'Judith', 'Halifax',6,1),
          (CURRENT DATE,'Marieke', 'Medway',5,3),
          (CURRENT DATE,'Hanneke', 'Halifax',5,2))
         ORDER BY sortkey

SALES_DATE SALES_PERSON    REGION          SALES       SORTKEY
---------- --------------- --------------- ----------- -----------
07/16/2003 Judith          Amsterdam                 6           1
07/16/2003 Hanneke         Halifax                   5           2
07/16/2003 Marieke         Medway                    5           3
```

You can also assign values to an include column in an INSERT operation by using a fullselect.

Include columns in UPDATE and DELETE operations

To assign values to include columns in UPDATE or DELETE operations, use the SET clause. If no value is assigned to an include column in the SET clause of an UPDATE or a DELETE statement, a NULL value is returned for that column.

In UPDATE statements, you can use include columns to return both the old and new column values for a row. For example:

```
SELECT salary, oldSalary FROM FINAL TABLE
       (UPDATE employee INCLUDE (oldSalary decimal(9,2))
        SET oldSalary = salary, salary = salary * 1.05
        WHERE job = 'CLERK')

SALARY      OLDSALARY
----------- -----------
   30712.50    29250.00
   23289.00    22180.00
   30198.00    28760.00
   20139.00    19180.00
   18112.50    17250.00
   28749.00    27380.00
```

Related concepts:

- "Retrieval of result sets from an SQL data change statement" on page 671

- "Retrieval of result sets from SQL data change statements using cursors" on page 674

Related reference:

- "DELETE statement" in the *SQL Reference, Volume 2*
- "INSERT statement" in the *SQL Reference, Volume 2*
- "SELECT statement" in the *SQL Reference, Volume 2*
- "UPDATE statement" in the *SQL Reference, Volume 2*

Searched UPDATE, INSERT, DELETE, and MERGE operations against fullselects

As of DB2® Version 8.1.4 you can issue searched INSERT, UPDATE, DELETE, and MERGE statements on the results of fullselects. This feature enables you to reduce work that might otherwise require two statements (a fullselect and an INSERT, UPDATE, DELETE, or MERGE on the results of the fullselect) to a single statement. By combining this work into a single statement, you can reduce the potential for deadlocks and possibly eliminate the need for view and cursor definitions. Any query that can be used to produce an insertable, updatable, or deletable view can be the target of searched INSERT, UPDATE, DELETE, or MERGE statements.

For example, the following statement will delete the ten employees in the EMPLOYEE table with the lowest level of education.

```
DELETE FROM (SELECT edlevel FROM employee
               ORDER BY edlevel
               FETCH FIRST 10 ROWS ONLY)
```

Related reference:

- "DELETE statement" in the *SQL Reference, Volume 2*
- "INSERT statement" in the *SQL Reference, Volume 2*
- "SELECT statement" in the *SQL Reference, Volume 2*
- "SELECT INTO statement" in the *SQL Reference, Volume 2*
- "UPDATE statement" in the *SQL Reference, Volume 2*
- "MERGE statement" in the *SQL Reference, Volume 2*

Sequential Values and Sequence Objects

The sections that follow describe considerations for sequential values and sequence objects.

Generation of Sequential Values

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

PREVAL expression in that statement. The NEXTVAL sequence expression increments the value of the sequence object and returns the new value of the sequence object.

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

```
CREATE SEQUENCE id_values
```

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

```
VALUES NEXTVAL FOR id_values

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

            1 record(s) selected.
```

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

```
VALUES PREVVAL FOR id_values

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

            1 record(s) selected.
```

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

```
VALUES PREVVAL FOR id_values

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

            1 record(s) selected.

VALUES PREVVAL FOR id_values

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

            1 record(s) selected.

VALUES NEXTVAL FOR id_values

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

            1 record(s) selected.

VALUES PREVVAL FOR id_values

1
```

```
-----------
          2

            1 record(s) selected.
```

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

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

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

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

Related reference:

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

Related samples:

- "DbSeq.java -- How to create, alter and drop a sequence in a database (JDBC)"

Management of Sequence Behavior

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

Data type
: The AS clause of the CREATE SEQUENCE statement specifies the numeric data type of the sequence object. The data type determines the possible minimum and maximum values of the sequence object (the minimum and maximum values for a data type are listed in the topic describing SQL limits). You cannot change the data type of a sequence object; instead, you must drop the sequence object by issuing the DROP SEQUENCE statement and issue a CREATE SEQUENCE statement with the new data type.

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

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

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

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

Sequence cycling
: The CYCLE clause causes the value of a sequence object that reaches its

maximum or minimum value to generate its respective minimum value or maximum value on the following NEXTVAL expression.

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

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

Related reference:

- "SQL limits" in the *SQL Reference, Volume 1*
- "ALTER SEQUENCE statement" in the *SQL Reference, Volume 2*
- "CREATE SEQUENCE statement" in the *SQL Reference, Volume 2*

Application Performance and Sequence Objects

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

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

If your sequence object must generate values in order, without introducing gaps in that order because of a system failure or database deactivation, use the ORDER and NO CACHE clauses in the CREATE SEQUENCE statement. The NO CACHE clause guarantees that no gaps appear in the generated values at the cost of some of your application's performance because it forces your sequence object to write to the database log every time it generates a new value. Note that gaps can still appear due to transactions that rollback and do not actually use that sequence value that they requested.

Sequence Objects Compared to Identity Columns

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

Declared Temporary Tables and Application Performance

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

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

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

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

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

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

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

```
DECLARE GLOBAL TEMPORARY TABLE TT1
```

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

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

Note that DB2 also enables you to create persistent tables with the SESSION schema. If you create a persistent table with the qualified name SESSION.TT3, you can then create a declared temporary table with the qualified name SESSION.TT3. In this situation, DB2 always resolves references to persistent and declared temporary tables with identical qualified names to the declared temporary table. To

avoid confusing persistent tables with declared temporary tables, you should not create persistent tables using the SESSION schema.

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

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

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

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

- If the table was created without the NOT LOGGED parameter (that is, the table is logged), only the changes made by the failed INSERT, UPDATE, or DELETE statement are rolled back.
- If the table was created with the NOT LOGGED parameter, DB2 deletes all of the rows of the declared temporary table.

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

Declared temporary tables are subject to a number of restrictions. For example, you cannot define aliases or views for declared temporary tables. You cannot use IMPORT and LOAD to populate declared temporary tables. You can, with some restrictions, create indexes for declared temporary tables. In addition, you can execute RUNSTATS against a declared temporary table to update the statistics for the declared temporary table and its indexes.

Related reference:

- "DECLARE GLOBAL TEMPORARY TABLE statement" in the *SQL Reference, Volume 2*

Related samples:

- "tbtemp.sqc -- How to use a declared temporary table (C)"
- "TbTemp.java -- How to use Declared Temporary Table (JDBC)"

Transmission of Large Volumes of Data Across a Network

You can combine the techniques of stored procedures and row blocking to significantly improve the performance of applications that need to pass large amounts of data across a network.

Applications that pass arrays, large amounts of data, or packages of data across the network can pass the data in blocks using the SQLDA data structure or host variables as the transport mechanism. This technique is extremely powerful in host languages that support structures.

Either a client application or a server procedure can pass the data across the network. The data can be passed using one of the following data types:

- VARCHAR
- LONG VARCHAR
- CLOB
- BLOB

The data can also be passed using one of the following graphic types:

- VARGRAPHIC
- LONG VARGRAPHIC
- DBCLOB

Note: Be sure to consider the possibility of character conversion when using this technique. If you are passing data with one of the character string data types such as VARCHAR, LONG VARCHAR, or CLOB, or graphic data types such as VARGRAPHIC, LONG VARGRAPHIC, OR DBCLOB, and the application code page is not the same as the database code page, any non-character data will be converted as if it were character data. To avoid character conversion, you should pass data in a variable with a data type of BLOB.

Related concepts:

- "Character conversion between different code pages" on page 609
- "DB2 Stored Procedures" on page 18

Related tasks:

- "Specifying row blocking to reduce overhead" in the *Administration Guide: Performance*

Part 8. Appendixes

Appendix A. Supported SQL Statements

The following table lists all the supported SQL statements in DB2 Universal Database for Linux, UNIX, and Windows. All of the statements in the ″SQL Statement″ column are supported in static SQL applications. The remaining columns show whether individual statements are supported in other DB2 application development contexts.

Table 93. SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Triggers[4]	SQL User-defined functions (UDFs) and methods	SQL Procedures
ALLOCATE CURSOR					X
ALTER { BUFFERPOOL, DATABASE PARTITION GROUP, FUNCTION, METHOD, NICKNAME, [8] PROCEDURE, SEQUENCE, SERVER, [8] TABLE, TABLESPACE, TYPE, USER MAPPING, [8] VIEW }	X				
ASSOCIATE LOCATORS					X
BEGIN DECLARE SECTION[2]					
CALL		X[7]	X	X	X
CASE statement					X
CLOSE		X			X
COMMENT ON	X	X			X
COMMIT	X	X			X
Compound SQL (Embedded)					
compound statement					X
CONNECT (Type 1)		X			
CONNECT (Type 2)		X			
CREATE { ALIAS, BUFFERPOOL, DATABASE PARTITION GROUP, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING,[8] INDEX, INDEX EXTENSION, METHOD, NICKNAME,[8] PROCEDURE, SCHEMA, SEQUENCE, SERVER, TABLE, TABLESPACE, TRANSFORM, TRIGGER, TYPE, TYPE MAPPING,[8] USER MAPPING,[8] VIEW, WRAPPER[8] }	X	X			X[9]
DECLARE CURSOR[2]		X			X
DECLARE GLOBAL TEMPORARY TABLE	X	X			X
DELETE	X	X	X[3]	X	X
DESCRIBE[6]		X			
DISCONNECT		X			

Table 93. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Triggers[4]	SQL User-defined functions (UDFs) and methods	SQL Procedures
DROP	X	X			X[9]
END DECLARE SECTION[2]					
EXECUTE					X
EXECUTE IMMEDIATE					X
EXPLAIN	X	X			X
FETCH		X			X
FLUSH EVENT MONITOR	X	X			
FLUSH PACKAGE CACHE	X	X			X
FOR statement			X	X	X
FREE LOCATOR					
GET DIAGNOSTICS			X	X	X
GOTO statement					X
GRANT	X	X			X
IF statement			X	X	X
INCLUDE[2]					
INSERT	X	X	X[3]	X	X
ITERATE			X	X	X
LEAVE statement			X	X	X
LOCK TABLE	X	X			X
LOOP statement					X
MERGE	X	X	X[3]	X	X
OPEN		X			X
PREPARE					X
REFRESH TABLE	X	X			
RELEASE		X			
RELEASE SAVEPOINT	X	X			X
RENAME TABLE	X	X			
RENAME TABLESPACE	X	X			
REPEAT statement					X
RESIGNAL statement					X
RETURN statement				X	X
REVOKE	X	X			
ROLLBACK	X	X			X
SAVEPOINT	X	X			X
select-statement	X	X	X	X	
SELECT INTO					X
SET CONNECTION		X			

Table 93. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Triggers[4]	SQL User-defined functions (UDFs) and methods	SQL Procedures
SET CURRENT DEFAULT TRANSFORM GROUP	X	X			X
SET CURRENT DEGREE	X	X			X
SET CURRENT EXPLAIN MODE	X	X			X
SET CURRENT EXPLAIN SNAPSHOT	X	X			X
SET CURRENT ISOLATION	X	X			X
SET CURRENT LOCK TIMEOUT	X	X			X
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	X	X			X
SET CURRENT PACKAGE PATH					
SET CURRENT PACKAGESET					
SET CURRENT QUERY OPTIMIZATION	X	X			X
SET CURRENT REFRESH AGE	X	X			X
SET ENCRYPTION PASSWORD	X	X			X
SET EVENT MONITOR STATE	X	X			X
SET INTEGRITY	X	X			
SET PASSTHRU[8]	X	X			
SET PATH	X	X			X
SET SCHEMA	X	X			X
SET SERVER OPTION[8]	X	X			
SET SESSION AUTHORIZATION[8]	X	X			
Set variable			X	X	X
SIGNAL statement			X	X	X
SIGNAL SQLSTATE[5]	X	X			
UPDATE	X	X	X[3]	X	X
VALUES INTO					X
WHENEVER[2]					
WHILE statement			X	X	X

Table 93. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Triggers[4]	SQL User-defined functions (UDFs) and methods	SQL Procedures

Notes:

1. You can code all statements in this list as static SQL, but only those marked with X as dynamic SQL.
2. You cannot execute this statement.
3. You cannot modify table data in before-triggers. Therefore, you cannot CALL procedures defined with MODIFIES SQL DATA or use INSERT, UPDATE, DELETE, or MERGE statements in before-triggers.
4. SQL statements in triggers cannot reference undefined transition variables, federated objects, or declared temporary tables. Also, SQL statements in before-triggers cannot reference materialized query tables defined with REFRESH IMMEDIATE.

   For a complete list of the restrictions for triggers, see "CREATE TRIGGER statement" in the *SQL Reference, Volume 2* .
5. You can only use this statement within CREATE FUNCTION, CREATE METHOD, CREATE PROCEDURE, or CREATE TRIGGER statements.
6. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command.
7. When CALL is issued through the command line processor, only certain procedures and their respective parameters are supported.
8. Statement is supported only for federated database servers.
9. SQL procedures can only issue CREATE and DROP statements for indexes, tables, and views.

Related reference:

- "DESCRIBE statement" in the *SQL Reference, Volume 2*
- "JAR file administration on the database server" in the *Application Development Guide: Programming Server Applications*

Appendix B. Security plug-in deployment limitations

The following are limitations on the use of security plug-ins:

DB2 Universal JDBC Driver support limitations:

The DB2® Universal JDBC Driver does not support the client side plug-in authentication model. Therefore you will not be able to use a GSS-API authentication plug-in to connect to a DB2 Universal Database (DB2 UDB) for Linux, UNIX®, and Windows® server from a DB2 Universal JDBC Driver client. A DB2 Universal JDBC Driver client can only use the supported operating system level authentication mechanism or the Kerberos authentication method. This limitation applies to both Type 2 and Type 4 connectivity.

Specifically, the server's database manager configuration parameter *srvcon_auth* cannot be set to GSSPLUGIN if simultaneously the database manager configuration parameter *srvcon_gssplugin_list* value does not contain the name of a Kerberos based GSS-API plug-in.

The *srvcon_auth* parameter can however be set to any of: CLIENT, SERVER, SERVER_ENCRYPT, KERBEROS, KRB_SERVER_ENCRYPT, or GSS_SERVER_ENCRYPT, DATA_ENCRYPT, or DATA_ENCRYPT_CMP.

DB2 UDB family support limitations:

You cannot use GSS-API plug-in to authenticate connections between a DB2 UDB for Linux, UNIX, and Windows client and another DB2 UDB family server. You also cannot authenticate connections from another DB2 UDB family server, acting as a client, to a DB2 UDB for Linux, UNIX, and Windows server.

If you use a DB2 UDB for Linux, UNIX, and Windows client to connect to other DB2 UDB family servers, you can however use client-side user ID/password plug-ins such as the IBM®-shipped operating system authentication plug-in, or you can write your own user ID/password plug-in. You can also use Kerberos plug-ins, or you can implement your own Kerberos plug-in.

With a DB2 UDB for Linux, UNIX, and Windows client you should not catalog a database using the GSSPLUGIN authentication type.

DB2 Information Integrator support limitations:

DB2 II does not support the use of delegated credentials from a GSS_API plug-in to establish outbound connections to data sources. Connections to data sources must continue to use the CREATE USER MAPPING command.

Database Administration Server support limitations:

The DB2 Administration Server (DAS) does not support security plug-ins. DAS only supports the operating system authentication mechanism.

Appendix C. Programming in a Host or iSeries Environment

Applications in Host or iSeries Environments

DB2® Connect lets an application program access data in DB2 databases on System/390®, zSeries®, iSeries™ servers. For example, an application running on Windows® can access data in a DB2 Universal Database for z/OS and OS/390 database. You can create new applications, or modify existing applications to run in a host or iSeries environment. You can also develop applications in one environment and port them to another.

DB2 Connect™ enables you to use the following APIs with host database products such as DB2 Universal Database for z/OS and OS/390, as long as the item is supported by the host database product:

- Embedded SQL, both static and dynamic
- The DB2 Call Level Interface
- The Microsoft® ODBC API
- JDBC

Some SQL statements differ among relational database products. You may encounter SQL statements that are:

- The same for all the database products that you use regardless of standards
- Available in all IBM® relational database products (see your SQL reference information for details)
- Unique to one database system that you access.

SQL statements in the first two categories are highly portable, but those in the third category will first require changes. In general, SQL statements in Data Definition Language (DDL) are not as portable as those in Data Manipulation Language (DML).

DB2 Connect accepts some SQL statements that are not supported by DB2 Universal Database. DB2 Connect passes these statements on to the host or iSeries

server. For information on limits on different platforms, such as the maximum column length, see the topic on SQL limits.

If you move a CICS® application from OS/390® or VSE to run under another CICS product (for example, CICS for AIX), it can also access the OS/390 or VSE database using DB2 Connect. Refer to the *CICS/6000 Application Programming Guide* and the *CICS Customization and Operation* manual for more details.

Note: You can use DB2 Connect with a DB2 Universal Database Version 8 database, although all you need is a DB2 client. Most of the incompatibility issues listed in the following topics will not apply if you are using DB2 Connect against a DB2 Universal Database Version 8 database, except in cases where a restriction is due to a limitation of DB2 Connect itself.

Related tasks:

- "Creating the sample database on Host or AS/400 and iSeries servers" in the *Application Development Guide: Building and Running Applications*

Related reference:

- "SQL limits" in the *SQL Reference, Volume 1*

Data Definition Language in Host and iSeries Environments

DDL statements differ among the IBM® database products because storage is handled differently on different systems. On host or iSeries™ server systems, there can be several steps between designing a database and issuing a CREATE TABLE statement. For example, a series of statements may translate the design of logical objects into the physical representation of those objects in storage.

The precompiler passes many such DDL statements to the host or iSeries server when you precompile to a host or iSeries server database. The same statements would not precompile against a database on the system where the application is running. For example, in an Windows® application the CREATE STORGROUP statement will precompile successfully to a DB2 Universal Database for z/OS and OS/390 database, but not to a DB2® for Windows database.

Data Manipulation Language in Host and iSeries Environments

In general, DML statements are highly portable. SELECT, INSERT, UPDATE, and DELETE statements are similar across the IBM® relational database products. Most applications primarily use DML SQL statements, which are supported by DB2® Connect.

Following are the considerations for using DML in host and iSeries™ environments:

- Numeric™ data types

  When numeric data is transferred to DB2 Universal Database, the data type may change. Numeric and zoned decimal SQLTYPE values, supported by OS/400®, are converted to fixed (packed) decimal SQLTYPE values.
- Mixed-byte data

  Mixed-byte data can consist of characters from an extended UNIX® code (EUC) character set, a double-byte character set (DBCS) and a single-byte character set (SBCS) in the same column. On systems that store data in EBCDIC (OS/390, z/OS™, OS/400, VSE, and VM), shift-out and shift-in characters mark the start

and end of double-byte data. On systems that store data in ASCII (such as UNIX), shift-in and shift-out characters are not required.

If your application transfers mixed-byte data from an ASCII system to an EBCDIC system, be sure to allow enough room for the shift characters. For each switch from SBCS to DBCS data, add 2 bytes to your data length. For better portability, use variable-length strings in applications that use mixed-byte data.

- Long fields

  Long fields (strings longer than 254 characters) are handled differently on different systems. A host or iSeries server may support only a subset of scalar functions for long fields; for example, DB2 Universal Database for z/OS and OS/390 allows only the **LENGTH** and **SUBSTR** functions for long fields. Also, a host or iSeries server may require different handling for certain SQL statements; for example, DB2 Server for VSE & VM requires that with the INSERT statement, only a host variable, the SQLDA, or a NULL value be used.

- Large object data type

  The LOB data type is supported by DB2 Connect.

- User-defined types

  Only user-defined distinct types are supported by DB2 Connect. Structured types, also known as abstract data types, are not supported by DB2 Connect.

- ROWID data type

  The ROWID data type is handled by DB2 Connect as VARCHAR for bit data.

- BIGINT data type

  Eight byte (64-bit) integers are supported by DB2 Connect. The BIGINT internal data type is used to provide support for the cardinality of very large databases, while retaining data precision.

Data Control Language in Host and iSeries Environments

Each IBM® relational database management system provides different levels of granularity for the GRANT and REVOKE SQL statements. Check the product-specific publications to verify the appropriate SQL statements to use for each database management system.

Database Connection Management with DB2 Connect

DB2® Connect supports the CONNECT TO and CONNECT RESET versions of the CONNECT statement, as well as CONNECT with no parameters. If an application calls an SQL statement without first performing an explicit CONNECT TO statement, an *implicit* connect is performed to the default application server (if one is defined).

When you connect to a database, information identifying the relational database management system is returned in the SQLERRP field of the SQLCA. If the application server is an IBM® relational database, the first three bytes of SQLERRP contain one of the following:

DSN DB2 Universal Database for z/OS and OS/390

ARI DB2 Server for VSE & VM

QSQ DB2 UDB for iSeries™

SQL DB2 Universal Database.

If you issue a CONNECT TO or null CONNECT statement while using DB2 Connect™, the territory code or territory token in the SQLERRMC field of the SQLCA is returned as blanks; the CCSID of the application server is returned in the code page or code set token.

You can explicitly disconnect by using the CONNECT RESET statement (for type 1 connect), the RELEASE and COMMIT statements (for type 2 connect), or the DISCONNECT statement (either type of connect, but not in a TP monitor environment).

Note: An application can receive SQLCODE values indicating errors and still end normally; DB2 Connect commits the data in this case. If you do not want the data to be committed, you must issue a ROLLBACK command.

The FORCE command lets you disconnect selected users or all users from the database. This is supported for host and iSeries server databases; the user can be forced off the DB2 Connect workstation.

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*

Processing of Interrupt Requests

DB2® Connect handles an interrupt request from a DB2 client in one of two ways:

- If the keyword INTERRUPT_ENABLED exists in the PARMS field of the DCS catalog entry, DB2 Connect™ will drop the connection to the host or iSeries™ server on receipt of an interrupt request. The loss of connection, at least on DB2 UDB for OS/390® and z/OS™ servers, will cause the current request to be interrupted at the server.
- If the keyword INTERRUPT_ENABLED does not exist in the PARMS field of the DCS catalog entry, interrupt requests are ignored.

Package Attributes, PREP, and BIND

The sections that follow describe differences in package attributes across IBM relational database systems, and considerations for the PREPCOMPILE and BIND commands.

Package Attribute Differences among IBM Relational Database Systems

A package has the following attributes:

Collection ID
The ID of the package. It can be specified on the PREP command.

Owner
The authorization ID of the package owner. It can be specified on the PREP or BIND command.

Creator
The user name that binds the package.

Qualifier
The implicit qualifier for objects in the package. It can be specified on the PREP or BIND command.

Each host or iSeries™ server system has limitations on the use of these attributes:

DB2 Universal Database for z/OS and OS/390
All four attributes can be different. The use of a different qualifier requires special administrative privileges. For more information on the conditions concerning the usage of these attributes, refer to the *Command Reference* for DB2 Universal Database for z/OS and OS/390.

DB2 Server for VSE & VM
All of the attributes must be identical. If USER1 creates a bind file (with PREP), and USER2 performs the actual bind, USER2 needs DBA authority to bind for USER1. Only USER1's user name is used for the attributes.

DB2® UDB for iSeries
The qualifier indicates the collection name. The relationship between qualifiers and ownership affects the granting and revoking of privileges on the object. The user name that is logged on is the creator and owner unless it is qualified by a collection ID, in which case the collection ID is the owner. The collection ID must already exist before it is used as a qualifier.

DB2 Universal Database
All four attributes can be different. The use of a different owner requires administrative authority and the binder must have CREATEIN privilege on the schema (if it already exists).

CNULREQD BIND Option for C Null-Terminated Strings

The CNULREQD bind option overrides the handling of null-terminated strings that are specified using the LANGLEVEL option.

By default, CNULREQD is set to YES. This causes null-terminated strings to be interpreted according to MIA standards. If connecting to a DB2 Universal Database for z/OS and OS/390 server, it is strongly recommended that you set CNULREQD to YES. You need to bind applications coded to SAA1 standards (with respect to null-terminated strings) with the CNULREQD option set to NO. Otherwise, null-terminated strings will be interpreted according to MIA standards, even if they are prepared using LANGLEVEL set to SAA1.

Related concepts:

- "Null-Terminated Strings in C and C++" on page 153

Standalone SQLCODE and SQLSTATE Variables

Standalone SQLCODE and SQLSTATE variables, as defined in ISO/ANS SQL92, are supported through the LANGLEVEL SQL92E precompile option. An SQL0020W warning will be issued at precompile time, indicating that LANGLEVEL is not supported. This warning applies only to the features listed under LANGLEVEL MIA, which is a subset of LANGLEVEL SQL92E.

Related reference:

- "PRECOMPILE Command" in the *Command Reference*

Isolation Levels Supported by DB2 Connect

DB2 Connect accepts the following isolation levels when you prep or bind an application:

RR Repeatable Read

RS Read Stability

CS Cursor Stability

UR Uncommitted Read

NC No Commit

The isolation levels are listed in order from most protection to least protection. If the host or iSeries™ server does not support the isolation level that you specify, the next higher supported level is used.

The following table shows the result of each isolation level on each host or iSeries application server.

Table 94. Isolation Levels

DB2 Connect	DB2 Universal Database for z/OS and OS/390	DB2 Server for VSE & VM	DB2® UDB for iSeries	DB2 Universal Database
RR	RR	RR	note 1	RR
RS	note 2	RR	COMMIT(*ALL)	RS
CS	CS	CS	COMMIT(*CS)	CS
UR	note 3	CS	COMMIT(*CHG)	UR
NC	note 4	note 5	COMMIT(*NONE)	UR

Notes:

1. There is no equivalent COMMIT option on DB2 UDB for iSeries that matches RR. DB2 UDB for iSeries support RR by locking the whole table.
2. Results in RR for Version 3.1, and results in RS for Version 4.1 with APAR PN75407 or Version 5.1.
3. Results in CS for Version 3.1, and results in UR for Version 4.1 or Version 5.1.
4. Results in CS for Version 3.1, and results in UR for Version 4.1 with APAR PN60988 or Version 5.1.
5. Isolation level NC is not supported with DB2 Server for VSE & VM.

With DB2 UDB for iSeries, you can access an unjournalled table if an application is bound with an isolation level of UR and blocking set to ALL, or if the isolation level is set to NC.

User-Defined Sort Orders

The differences between EBCDIC and ASCII cause differences in sort orders in the various database products, and also affect ORDER BY and GROUP BY clauses. One way to minimize these differences is to create a user-defined collating sequence that mimics the EBCDIC sort order. You can specify a collating sequence only when you create a new database.

Note: Database tables can now be stored on DB2 Universal Database for z/OS and OS/390 in ASCII format. This permits faster exchange of data between DB2

Connect and DB2 Universal Database for z/OS and OS/390, and removes the need to provide field procedures which must otherwise be used to convert data and resequence it.

Referential Integrity Differences among IBM Relational Database Systems

Different systems handle referential constraints differently:

DB2 Universal Database for z/OS and OS/390
An index must be created on a primary key before a foreign key can be created using the primary key. Tables can reference themselves.

DB2 Server for VSE & VM
An index is automatically created for a foreign key. Tables cannot reference themselves.

DB2® UDB for iSeries™
An index is automatically created for a foreign key. Tables can reference themselves.

DB2 Universal Database
For DB2 Universal Database databases, an index is automatically created for a unique constraint, including a primary key. Tables can reference themselves.

Other rules vary concerning levels of cascade.

Locking and Application Portability

The way in which the database server performs locking can affect some applications. For example, applications designed around row-level locking and the isolation level of cursor stability are not directly portable to systems that perform page-level locking. Because of these underlying differences, applications may need to be adjusted.

The DB2 Universal Database for z/OS and OS/390 and DB2 Universal Database products have the ability to time-out a lock and send an error return code to waiting applications.

SQLCODE and SQLSTATE Differences among IBM Relational Database Systems

Different IBM® relational database products do not always produce the same SQLCODE values for similar errors. You can handle this problem in either of two ways:

- Use the SQLSTATE instead of the SQLCODE for a particular error.

  SQLSTATE values have approximately the same meaning across the database products, and the products produce SQLSTATE values that correspond to the SQLCODE values.
- Map the SQLCODE values from one system to another system.

  By default, DB2® Connect maps SQLCODE values and tokens from each IBM host or iSeries™ server system to your DB2 Universal Database system. You can specify your own SQLCODE mapping file if you want to override the default

mapping or you are using a database server that does not have SQLCODE mapping (a non-IBM database server). You can also turn off SQLCODE mapping.

Related concepts:

- "SQLCODE mapping" in the *DB2 Connect User's Guide*

System Catalog Differences among IBM Relational Database Systems

The system catalogs vary across the IBM® database products. Many differences can be masked by the use of views. For information, see the documentation for the database server that you are using.

The catalog functions in CLI avoid this problem by presenting support of the same API and result sets for catalog queries across the DB2® family.

Related concepts:

- "Catalog functions for querying system catalog information in CLI applications" in the *CLI Guide and Reference, Volume 1*

Numeric Conversion Overflows on Retrieval Assignments

Numeric™ conversion overflows on retrieval assignments may be handled differently by different IBM® relational database products. For example, consider fetching a float column into an integer host variable from DB2 Universal Database for z/OS and OS/390 and from DB2 Universal Database. When converting the float value to an integer value, a conversion overflow may occur. By default, DB2 Universal Database for z/OS and OS/390 will return a warning SQLCODE and a null value to the application. In contrast, DB2 Universal Database will return a conversion overflow error. It is recommended that applications avoid numeric conversion overflows on retrieval assignments by fetching into appropriately sized host variables.

Stored Procedures in Host or iSeries Environments

The considerations for stored procedures in host and iSeries™ environments are as follows:

- Invocation

  A client program can invoke a server program by issuing an SQL CALL statement. Each server works a little differently to the other servers in this case.

  z/OS™ and OS/390®
  : The schema name must be no more than 8 bytes long, the procedure name must be no more than 18 bytes long, and the stored procedure must be defined in the SYSIBM.SYSPROCEDURES catalog on the server.

  VSE or VM
  : The procedure name must not be more than 18 bytes long and must be defined in the SYSTEM.SYSROUTINES catalog on the server.

  OS/400®
  : The procedure name must be an SQL identifier. You can also use the

DECLARE PROCEDURE or CREATE PROCEDURE statements to specify the actual path name (the schema-name or collection-name) to locate the stored procedure.

All CALL statements to DB2® UDB for iSeries from REXX/SQL must be dynamically prepared and executed by the application, as the CALL statement implemented in REXX/SQL maps to CALL USING DESCRIPTOR.

You can invoke the server program on DB2 Universal Database with the same parameter convention that server programs use on DB2 Universal Database for z/OS and OS/390, DB2 UDB for iSeries or DB2 Server for VSE & VM. For more information on the parameter convention on other platforms, refer to the DB2 product documentation for that platform.

All the SQL statements in a stored procedure are executed as part of the SQL unit of work started by the client SQL program.

- Do not pass indicator values with special meaning to or from stored procedures.

  Between DB2 Universal Database, the systems pass whatever you put into the indicator variables. However, when using DB2 Connect™, you can only pass 0, -1, and -128 in the indicator variables.
- You should define a parameter to return any error or warning encountered by the server application.

  A server program on DB2 Universal Database can update the SQLCA to return any error or warning, but a stored procedure on DB2 Universal Database for z/OS and OS/390 or DB2 UDB for iSeries has no such support. If you want to return an error code from your stored procedure, you must pass it as a parameter. The SQLCODE and SQLCA is only set by the server for system detected errors.
- DB2 Server for VSE & VM Version 7 or higher, DB2 Universal Database for z/OS and OS/390 Version 5.1 or higher, DB2 for AS/400® V5R1, and DB2 for iSeries Version 7 or higher are the only host or iSeries application servers that can return the result sets of stored procedures at this time.

Related concepts:

- "DB2 Stored Procedures" on page 18

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*

DB2 Connect Support for Compound SQL

Compound SQL allows multiple SQL statements to be grouped into a single executable block. This may reduce network overhead and improve response time.

With NOT ATOMIC compound SQL, processing of compound SQL continues following an error. With ATOMIC compound SQL, an error rolls back the entire group of compound SQL.

Statements will continue execution until terminated by the application server. In general, execution of the compound SQL statement will be stopped only in the case of serious errors.

NOT ATOMIC compound SQL can be used with all of the supported host or iSeries™ application servers. ATOMIC compound SQL can be used with supported host application servers.

If multiple SQL errors occur, the SQLSTATE values of the first seven failing statements are returned in the SQLERRMC field of the SQLCA with a message that multiple errors occurred.

Related reference:

- "SQLCA" in the *Administrative API Reference*

Multisite Update with DB2 Connect

DB2® Connect allows you to perform a multisite update, also known as two-phase commit. A multisite update is an update of multiple databases within a single distributed unit of work (DUOW). Whether you can use this capability depends on several factors:

- Your application program must be precompiled with the CONNECT 2 and SYNCPOINT TWOPHASE options.
- If you have SNA network connections, you can use two-phase commit support provided by the sync point manager (SPM) function of DB2 Connect™ Enterprise Edition on AIX®, and Windows® NT. This feature enables the following host database servers to participate in a distributed unit of work:
  - DB2 for AS/400® Version 3.1 or later
  - DB2 UDB for iSeries™ Version 5.1 or later
  - DB2 for OS/390® Version 5.1 or later
  - DB2 UDB for OS/390 and z/OS™ Version 7 or later
  - DB2 for VM & VSE Version V5.1 or later.

  The above is true for native DB2 UDB applications and applications coordinated by an external TP monitor such as IBM® TXSeries®, CICS® for Open Systems, BEA Tuxedo, Encina® Monitor, and Microsoft® Transaction Server.
- If you have TCP/IP network connections, then a DB2 for OS/390 V5.1 or later server can participate in a distributed unit of work. If the application is controlled by a Transaction Processing Monitor such as IBM TXSeries, CICS for Open Systems, Encina Monitor, or Microsoft Transaction Server, then you must use SPM.

  If a common DB2 Connect Enterprise Edition server is used by both native DB2 applications and TP monitor applications to access host data over TCP/IP connections, the sync point manager must be used.

  If a single DB2 Connect Enterprise Edition server is used to access host data using both SNA and TCP/IP network protocols and two-phase commit is required, you must use SPM. This is true for both DB2 applications and TP monitor applications.

Related concepts:

- "XA function supported by DB2 Universal Database" in the *Administration Guide: Planning*
- "Configuring DB2 Connect with an XA compliant transaction manager" in the *DB2 Connect User's Guide*

Related tasks:

- "Configuring BEA Tuxedo" in the *Administration Guide: Planning*
- "Updating host or iSeries database servers with an XA-compliant transaction manager" in the *Administration Guide: Planning*

Host and iSeries Server SQL Statements Supported by DB2 Connect

The following statements compile successfully for host and iSeries™ server processing, but not for processing with DB2 Universal Database systems:

- ACQUIRE
- DECLARE (modifier.(qualifier.)table_name TABLE ...
- LABEL ON

These statements are also supported by the command line processor.

The following statements are supported for host and iSeries server processing but are not added to the bind file or the package and are not supported by the command line processor:

- DESCRIBE statement_name INTO descriptor_name USING NAMES
- PREPARE statement_name INTO descriptor_name USING NAMES FROM ...

The precompiler makes the following assumptions:

- Host variables are input variables
- The statement is assigned a unique section number.

Host and iSeries Server SQL Statements Rejected by DB2 Connect

The following SQL statements are not supported by DB2® Connect and not supported by the command line processor:

- COMMIT WORK RELEASE
- DECLARE state_name, statement_name STATEMENT
- DESCRIBE statement_name INTO descriptor_name USING xxxx (where xxxx is ANY, BOTH, or LABELS)
- PREPARE statement_name INTO descriptor_name USING xxxx FROM :host_variable (where xxxx is ANY, BOTH, or LABELS)
- PUT ...
- ROLLBACK WORK RELEASE
- SET :host_variable = CURRENT ...

DB2 Server for VSE & VM extended dynamic SQL statements are rejected with -104 and syntax error SQLCODE values.

Appendix D. Simulation of EBCDIC Binary Collation

With DB2®, you can collate character strings according to a user-defined collating sequence. You can use this feature to simulate EBCDIC binary collation.

As an example of how to simulate EBCDIC collation, suppose you want to create an ASCII database with code page 850, but you also want the character strings to be collated as if the data actually resides in an EBCDIC database with code page 500. See figures below for the definitions of code page 500 and code page 850.

Consider the relative collation of four characters in a EBCDIC code page 500 database, when they are collated in binary:

Character	Code Page 500 Code Point
'a'	X'81'
'b'	X'82'
'A'	X'C1'
'B'	X'C2'

The code page 500 binary collation sequence (the desired sequence) is:

```
'a' < 'b' < 'A' < 'B'
```

If you create the database with ASCII code page 850, binary collation would yield:

Character	Code Page 850 Code Point
'a'	X'61'
'b'	X'62'
'A'	X'41'
'B'	X'42'

The code page 850 binary collation (which is not the desired sequence) is:

```
'A' < 'B' < 'a' < 'b'
```

To achieve the desired collation, you need to create your database with a user-defined collating sequence. A sample collating sequence for just this purpose is supplied with DB2 in the `sqle850a.h` include file. The content of `sqle850a.h` is shown in the following.

```
#ifndef SQL_H_SQLE850A
#define SQL_H_SQLE850A

#ifdef __cplusplus
extern "C" {
#endif

unsigned char sqle_850_500[256] = {
0x00,0x01,0x02,0x03,0x37,0x2d,0x2e,0x2f,0x16,0x05,0x25,0x0b,0x0c,0x0d,0x0e,0x0f,
0x10,0x11,0x12,0x13,0x3c,0x3d,0x32,0x26,0x18,0x19,0x3f,0x27,0x1c,0x1d,0x1e,0x1f,
0x40,0x4f,0x7f,0x7b,0x5b,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
0x7c, 0xc1 , 0xc2 ,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,
0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0xe0,0x5a,0x5f,0x6d,
0x79, 0x81 , 0x82 ,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,0x95,0x96,
0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0xbb,0xd0,0xa1,0x07,
0x68,0xdc,0x51,0x42,0x43,0x44,0x47,0x48,0x52,0x53,0x54,0x57,0x56,0x58,0x63,0x67,
0x71,0x9c,0x9e,0xcb,0xcc,0xcd,0xdb,0xdd,0xdf,0xec,0xfc,0x70,0xb1,0x80,0xbf,0xff,
0x45,0x55,0xce,0xde,0x49,0x69,0x9a,0x9b,0xab,0xaf,0xba,0xb8,0xb7,0xaa,0x8a,0x8b,
0x2b,0x2c,0x09,0x21,0x28,0x65,0x62,0x64,0xb4,0x38,0x31,0x34,0x33,0xb0,0xb2,0x24,
0x22,0x17,0x29,0x06,0x20,0x2a,0x46,0x66,0x1a,0x35,0x08,0x39,0x36,0x30,0x3a,0x9f,
0x8c,0xac,0x72,0x73,0x74,0x0a,0x75,0x76,0x77,0x23,0x15,0x14,0x04,0x6a,0x78,0x3b,
0xee,0x59,0xeb,0xed,0xcf,0xef,0xa0,0x8e,0xae,0xfe,0xfb,0xfd,0x8d,0xad,0xbc,0xbe,
0xca,0x8f,0x1b,0xb9,0xb6,0xb5,0xe1,0x9d,0x90,0xbd,0xb3,0xda,0xfa,0xea,0x3e,0x41
};
#ifdef __cplusplus
}
#endif

#endif /* SQL_H_SQLE850A */
```

Figure 66. User-Defined Collating Sequence - sqle_850_500

To see how to achieve code page 500 binary collation on code page 850 characters, examine the sample collating sequence in `sqle_850_500`. For each code page 850 character, its weight in the collating sequence is simply its corresponding code point in code page 500.

For example, consider the letter 'a'. This letter is code point X'61' for code page 850. In the array `sqle_850_500`, letter 'a' is assigned a weight of X'81' (that is, the 98th element in the array `sqle_850_500`).

Consider how the four characters collate when the database is created with the above sample user-defined collating sequence:

Character	**Code Page 850 Code Point / Weight (from sqle_850_500)**
'a'	X'61' / X'81'
'b'	X'62' / X'82'
'A'	X'41' / X'C1'
'B'	X'42' / X'C2'

The code page 850 user-defined collation by weight (the desired collation) is:

```
'a' < 'b' < 'A' < 'B'
```

In this example, you achieve the desired collation by specifying the correct weights to simulate the desired behavior.

Closely observing the actual collating sequence, notice that the sequence itself is merely a conversion table, where the source code page is the code page of the data base (850) and the target code page is the desired binary collating code page (500). Other sample collating sequences supplied by DB2 enable different conversions. If

a conversion table that you require is not supplied with DB2, additional conversion tables can be obtained from the IBM® publication, *Character Data Representation Architecture, Reference and Registry*, SC09-2190. You will find the additional conversion tables in a CD-ROM enclosed with that publication.

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP010000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¢ SC040000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	• SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	[SM060000	] SM080000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	1 ND011000	2 ND021000	3 ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	‗ SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	, SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 00500

Figure 67. Code Page 500

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	Ç LC420000	É LE120000	á LA110000	▒ SF140000	┐ SF020000	ð LD630000	Ó LO120000	(SHY) SP320000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	▓ SF150000	└ SF070000	Ð LD620000	ß LS610000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	▓ SF160000	┬ SF060000	Ê LE160000	Ô LO160000	‗ SM100000
-3	♥ SS020000	!! SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	┤ SF110000	├ SF080000	Ë LE180000	Ò LO140000	¾ NF050000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	ñ LN190000	╡ SF090000	─ SF100000	È LE140000	õ LO190000	¶ SM250000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	ò LO130000	Ñ LN200000	Á LA120000	┼ SF050000	1 LI610000	Õ LO200000	§ SM240000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	ª SM210000	Â LA160000	ã LA190000	Í LI120000	µ SM170000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ù LU130000	º SM200000	À LA140000	Ã LA200000	Î LI160000	þ LT630000	, SD410000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY170000	¿ SP160000	© SM520000	╚ SF380000	Ï LI180000	Þ LT640000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	® SM530000	╣ SF230000	╔ SF390000	┘ SF040000	Ú LU120000	¨ SD170000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	╩ SF240000	╦ SF400000	┌ SF010000	Û LU160000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	ï LI170000	ø LO610000	½ NF010000	╗ SF250000	╠ SF410000	▀ SF610000	Ù LU140000	1 ND011000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	l SM130000	î LI150000	£ SC020000	¼ NF040000	┘ SF260000	═ SF420000	■ SP570000	ý LY110000	3 ND031000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] LM080000	m LM010000	} SM140000	ì LI130000	Ø LO620000	¡ SP030000	¢ SC040000	╬ SF430000	¦ SM650000	Ý LY120000	2 ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	× SA070000	« SP170000	¥ SC050000	╧ SF440000	Ì LI140000	¯ SM150000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Å LA280000	ƒ SC070000	» SP180000	┐ SF030000	¤ SC010000	▬ SF600000	´ SD110000	(RSP) SP300000

Code Page 00850

Figure 68. Code Page 850

Related concepts:

- "Collating sequences" on page 597

Related reference:

- "sqlecrea - Create Database" in the *Administrative API Reference*

Appendix E. DB2 Universal Database technical information

DB2 documentation and help

DB2® technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- Downloadable PDF files, PDF files on CD, and printed books
  - Guides
  - Reference manuals
- Command line help
  - Command help
  - Message help
  - SQL state help
- Installed source code
  - Sample programs

You can access additional DB2 Universal Database™ technical information such as technotes, white papers, and Redbooks™ online at ibm.com®. Access the DB2 Information Management software library site at www.ibm.com/software/data/pubs/.

DB2 documentation updates

IBM® may periodically make documentation FixPaks and other documentation updates to the DB2 Information Center available. If you access the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/, you will always be viewing the most up-to-date information. If you have installed the DB2 Information Center locally, then you need to install any updates manually before you can view them. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* when new information becomes available.

The Information Center is updated more frequently than either the PDF or the hardcopy books. To get the most current DB2 technical information, install the documentation updates as they become available or go to the DB2 Information Center at the www.ibm.com site.

Related concepts:

- "CLI sample programs" in the *CLI Guide and Reference, Volume 1*
- "Java sample programs" in the *Application Development Guide: Building and Running Applications*
- "DB2 Information Center" on page 708

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 725

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Invoking message help from the command line processor" on page 726
- "Invoking command help from the command line processor" on page 727
- "Invoking SQL state help from the command line processor" on page 727

Related reference:

- "DB2 PDF and printed documentation" on page 719

DB2 Information Center

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 family products, including DB2 Universal Database™, DB2 Connect™, DB2 Information Integrator and DB2 Query Patroller™. The DB2 Information Center also contains information for major DB2 features and components including replication, data warehousing, and the DB2 extenders.

The DB2 Information Center has the following features if you view it in Mozilla 1.0 or later or Microsoft® Internet Explorer 5.5 or later. Some features require you to enable support for JavaScript™:

Flexible installation options

You can choose to view the DB2 documentation using the option that best meets your needs:

- To effortlessly ensure that your documentation is always up to date, you can access all of your documentation directly from the DB2 Information Center hosted on the IBM® Web site at http://publib.boulder.ibm.com/infocenter/db2help/
- To minimize your update efforts and keep your network traffic within your intranet, you can install the DB2 documentation on a single server on your intranet
- To maximize your flexibility and reduce your dependence on network connections, you can install the DB2 documentation on your own computer

Search

You can search all of the topics in the DB2 Information Center by entering a search term in the **Search** text field. You can retrieve exact matches by enclosing terms in quotation marks, and you can refine your search with wildcard operators (*, ?) and Boolean operators (AND, NOT, OR).

Task-oriented table of contents

You can locate topics in the DB2 documentation from a single table of contents. The table of contents is organized primarily by the kind of tasks you may want to perform, but also includes entries for product overviews, goals, reference information, an index, and a glossary.

- Product overviews describe the relationship between the available products in the DB2 family, the features offered by each of those products, and up to date release information for each of these products.
- Goal categories such as installing, administering, and developing include topics that enable you to quickly complete tasks and develop a deeper understanding of the background information for completing those tasks.

- Reference topics provide detailed information about a subject, including statement and command syntax, message help, and configuration parameters.

Show current topic in table of contents
You can show where the current topic fits into the table of contents by clicking the **Refresh / Show Current Topic** button in the table of contents frame or by clicking the **Show in Table of Contents** button in the content frame. This feature is helpful if you have followed several links to related topics in several files or arrived at a topic from search results.

Index You can access all of the documentation from the index. The index is organized in alphabetical order by index term.

Glossary
You can use the glossary to look up definitions of terms used in the DB2 documentation. The glossary is organized in alphabetical order by glossary term.

Integrated localized information
The DB2 Information Center displays information in the preferred language set in your browser preferences. If a topic is not available in your preferred language, the DB2 Information Center displays the English version of that topic.

For iSeries™ technical information, refer to the IBM eServer™ iSeries information center at www.ibm.com/eserver/iseries/infocenter/.

Related concepts:

- "DB2 Information Center installation scenarios" on page 709

Related tasks:

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Displaying topics in your preferred language in the DB2 Information Center" on page 718
- "Invoking the DB2 Information Center" on page 716
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 712
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 714

DB2 Information Center installation scenarios

Different working environments can pose different requirements for how to access DB2® information. The DB2 Information Center can be accessed on the IBM® Web site, on a server on your organization's network, or on a version installed on your computer. In all three cases, the documentation is contained in the DB2 Information Center, which is an architected web of topic-based information that you view with a browser. By default, DB2 products access the DB2 Information Center on the IBM Web site. However, if you want to access the DB2 Information Center on an intranet server or on your own computer, you must install the DB2 Information Center using the DB2 Information Center CD found in your product Media Pack. Refer to the summary of options for accessing DB2 documentation which follows, along with the three installation scenarios, to help determine which

method of accessing the DB2 Information Center works best for you and your work environment, and what installation issues you might need to consider.

Summary of options for accessing DB2 documentation:

The following table provides recommendations on which options are possible in your work environment for accessing the DB2 product documentation in the DB2 Information Center.

Internet access	Intranet access	Recommendation
Yes	Yes	Access the DB2 Information Center on the IBM Web site, or access the DB2 Information Center installed on an intranet server.
Yes	No	Access the DB2 Information Center on the IBM Web site.
No	Yes	Access the DB2 Information Center installed on an intranet server.
No	No	Access the DB2 Information Center on a local computer.

Scenario: Accessing the DB2 Information Center on your computer:

Tsu-Chen owns a factory in a small town that does not have a local ISP to provide him with Internet access. He purchased DB2 Universal Database™ to manage his inventory, his product orders, his banking account information, and his business expenses. Never having used a DB2 product before, Tsu-Chen needs to learn how to do so from the DB2 product documentation.

After installing DB2 Universal Database on his computer using the typical installation option, Tsu-Chen tries to access the DB2 documentation. However, his browser gives him an error message that the page he tried to open cannot be found. Tsu-Chen checks the installation manual for his DB2 product and discovers that he has to install the DB2 Information Center if he wants to access DB2 documentation on his computer. He finds the *DB2 Information Center CD* in the media pack and installs it.

From the application launcher for his operating system, Tsu-Chen now has access to the DB2 Information Center and can learn how to use his DB2 product to increase the success of his business.

Scenario: Accessing the DB2 Information Center on the IBM Web site:

Colin is an information technology consultant with a training firm. He specializes in database technology and SQL and gives seminars on these subjects to businesses all over North America using DB2 Universal Database. Part of Colin's seminars includes using DB2 documentation as a teaching tool. For example, while teaching courses on SQL, Colin uses the DB2 documentation on SQL as a way to teach basic and advanced syntax for database queries.

Most of the businesses at which Colin teaches have Internet access. This situation influenced Colin's decision to configure his mobile computer to access the DB2 Information Center on the IBM Web site when he installed the latest version of DB2 Universal Database. This configuration allows Colin to have online access to the latest DB2 documentation during his seminars.

However, sometimes while travelling Colin does not have Internet access. This posed a problem for him, especially when he needed to access to DB2 documentation to prepare for seminars. To avoid situations like this, Colin installed a copy of the DB2 Information Center on his mobile computer.

Colin enjoys the flexibility of always having a copy of DB2 documentation at his disposal. Using the **db2set** command, he can easily configure the registry variables on his mobile computer to access the DB2 Information Center on either the IBM Web site, or his mobile computer, depending on his situation.

Scenario: Accessing the DB2 Information Center on an intranet server:

Eva works as a senior database administrator for a life insurance company. Her administration responsibilities include installing and configuring the latest version of DB2 Universal Database on the company's UNIX® database servers. Her company recently informed its employees that, for security reasons, it would not provide them with Internet access at work. Because her company has a networked environment, Eva decides to install a copy of the DB2 Information Center on an intranet server so that all employees in the company who use the company's data warehouse on a regular basis (sales representatives, sales managers, and business analysts) have access to DB2 documentation.

Eva instructs her database team to install the latest version of DB2 Universal Database on all of the employee's computers using a response file, to ensure that each computer is configured to access the DB2 Information Center using the host name and the port number of the intranet server.

However, through a misunderstanding Migual, a junior database administrator on Eva's team, installs a copy of the DB2 Information Center on several of the employee computers, rather than configuring DB2 Universal Database to access the DB2 Information Center on the intranet server. To correct this situation Eva tells Migual to use the **db2set** command to change the DB2 Information Center registry variables (DB2_DOCHOST for the host name, and DB2_DOCPORT for the port number) on each of these computers. Now all of the appropriate computers on the network have access to the DB2 Information Center, and employees can find answers to their DB2 questions in the DB2 documentation.

Related concepts:

- "DB2 Information Center" on page 708

Related tasks:

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 712
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 714
- "Setting the location for accessing the DB2 Information Center: Common GUI help"

Related reference:

- "db2set - DB2 Profile Registry Command" in the *Command Reference*

Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a UNIX operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on UNIX computers.

- **Hardware requirements**

  You require one of the following processors:
  - PowerPC (AIX)
  - HP 9000 (HP-UX)
  - Intel 32–bit (Linux)
  - Solaris UltraSPARC computers (Solaris Operating Environment)
- **Operating system requirements**

  You require one of the following operating systems:
  - IBM AIX 5.1 (on PowerPC)
  - HP-UX 11i (on HP 9000)
  - Red Hat Linux 8.0 (on Intel 32–bit)
  - SuSE Linux 8.1 (on Intel 32–bit)
  - Sun Solaris Version 8 (on Solaris Operating Environment UltraSPARC computers)

  Note: The DB2 Information Center runs on a subset of the UNIX operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center from the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**
  - The following browser is supported:
    - Mozilla Version 1.0 or greater
- The DB2 Setup wizard is a graphical installer. You must have an implementation of the X Window System software capable of rendering a graphical user interface for the DB2 Setup wizard to run on your computer. Before you can run the DB2 Setup wizard you must ensure that you have properly exported your display. For example, enter the following command at the command prompt:

  ```
  export DISPLAY=9.26.163.144:0.
  ```
- **Communication requirements**
  - TCP/IP

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system.
2. Insert and mount the DB2 Information Center product CD on your system.
3. Change to the directory where the CD is mounted by entering the following command:

   ```
   cd /cd
   ```

   where */cd* represents the mount point of the CD.
4. Enter the **./db2setup** command to start the DB2 Setup wizard.
5. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
6. On the **Select the product you would like to install** page, click **Next**.
7. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
8. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
9. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
10. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
11. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
12. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can also install the DB2 Information Center using a response file.

The installation logs `db2setup.his`, `db2setup.log`, and `db2setup.err` are located, by default, in the `/tmp` directory.

The `db2setup.log` file captures all DB2 product installation information, including errors. The `db2setup.his` file records all DB2 product installations on your computer. DB2 appends the `db2setup.log` file to the `db2setup.his` file. The `db2setup.err` file captures any error output that is returned by Java, for example, exceptions and trap information.

When the installation is complete, the DB2 Information Center will be installed in one of the following directories, depending upon your UNIX operating system:

- AIX: /usr/opt/db2_08_01
- HP-UX: /opt/IBM/db2/V8.1
- Linux: /opt/IBM/db2/V8.1
- Solaris Operating Environment: /opt/IBM/db2/V8.1

Related concepts:

- "DB2 Information Center" on page 708
- "DB2 Information Center installation scenarios" on page 709

Related tasks:

- "Installing DB2 using a response file (UNIX)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Displaying topics in your preferred language in the DB2 Information Center" on page 718
- "Invoking the DB2 Information Center" on page 716
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 714

Installing the DB2 Information Center using the DB2 Setup wizard (Windows)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the DB2 documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a Windows operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on Windows.

- **Hardware requirements**

  You require one of the following processors:

  – 32-bit computers: a Pentium or Pentium compatible CPU
- **Operating system requirements**

  You require one of the following operating systems:

  – Windows 2000

  – Windows XP

  Note: The DB2 Information Center runs on a subset of the Windows operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center on the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**

  – The following browsers are supported:

    - Mozilla 1.0 or greater

    - Internet Explorer Version 5.5 or 6.0 (Version 6.0 for Windows XP)
- **Communication requirements**

  – TCP/IP

Restrictions:

- You require an account with administrative privileges to install the DB2 Information Center.

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system with the account that you have defined for the DB2 Information Center installation.
2. Insert the CD into the drive. If enabled, the auto-run feature starts the IBM DB2 Setup Launchpad.
3. The DB2 Setup wizard determines the system language and launches the setup program for that language. If you want to run the setup program in a language other than English, or the setup program fails to auto-start, you can start the DB2 Setup wizard manually.

   To start the DB2 Setup wizard manually:

   a. Click **Start** and select **Run**.
   b. In the **Open** field, type the following command:

      ```
      x:\setup.exe /i 2-letter language identifier
      ```

      where *x:* represents your CD drive, and *2-letter language identifier* represents the language in which the setup program will be run.
   c. Click **OK**.
4. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
5. On the **Select the product you would like to install** page, click **Next**.
6. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
7. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
8. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
9. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
10. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
11. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can install the DB2 Information Center using a response file. You can also use the **db2rspgn** command to generate a response file based on an existing installation.

For information on errors encountered during installation, see the `db2.log` and `db2wi.log` files located in the `'My Documents'\DB2LOG\` directory. The location of the `'My Documents'` directory will depend on the settings on your computer.

The `db2wi.log` file captures the most recent DB2 installation information. The `db2.log` captures the history of DB2 product installations.

Related concepts:

- "DB2 Information Center" on page 708
- "DB2 Information Center installation scenarios" on page 709

Related tasks:

- "Installing a DB2 product using a response file (Windows)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Displaying topics in your preferred language in the DB2 Information Center" on page 718
- "Invoking the DB2 Information Center" on page 716
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 712

Related reference:

- "db2rspgn - Response File Generator Command (Windows)" in the *Command Reference*

Invoking the DB2 Information Center

The DB2 Information Center gives you access to all of the information that you need to use DB2 products for Linux, UNIX, and Windows operating systems such as DB2 Universal Database, DB2 Connect, DB2 Information Integrator, and DB2 Query Patroller.

You can invoke the DB2 Information Center from one of the following places:

- Computers on which a DB2 UDB client or server is installed
- An intranet server or local computer on which the DB2 Information Center installed
- The IBM Web site

Prerequisites:

Before you invoke the DB2 Information Center:

- *Optional*: Configure your browser to display topics in your preferred language
- *Optional*: Configure your DB2 client to use the DB2 Information Center installed on your computer or intranet server

Procedure:

To invoke the DB2 Information Center on a computer on which a DB2 UDB client or server is installed:

- From the Start Menu (Windows operating system): Click **Start → Programs → IBM DB2 → Information → Information Center**.
- From the command line prompt:
  - For Linux and UNIX operating systems, issue the **db2icdocs** command.
  - For the Windows operating system, issue the **db2icdocs.exe** command.

To open the DB2 Information Center installed on an intranet server or local computer in a Web browser:

- Open the Web page at http://<host-name>:<port-number>/, where <host-name> represents the host name and <port-number> represents the port number on which the DB2 Information Center is available.

To open the DB2 Information Center on the IBM Web site in a Web browser:

- Open the Web page at publib.boulder.ibm.com/infocenter/db2help/.

Related concepts:

- "DB2 Information Center" on page 708
- "DB2 Information Center installation scenarios" on page 709

Related tasks:

- "Displaying topics in your preferred language in the DB2 Information Center" on page 718
- "Invoking contextual help from a DB2 tool" on page 725
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 717
- "Invoking command help from the command line processor" on page 727
- "Setting the location for accessing the DB2 Information Center: Common GUI help"

Related reference:

- "HELP Command" in the *Command Reference*

Updating the DB2 Information Center installed on your computer or intranet server

The DB2 Information Center available from http://publib.boulder.ibm.com/infocenter/db2help/ will be periodically updated with new or changed documentation. IBM may also make DB2 Information Center updates available to download and install on your computer or intranet server. Updating the DB2 Information Center does not update DB2 client or server products.

Prerequisites:

You must have access to a computer that is connected to the Internet.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:

1. Open the DB2 Information Center hosted on the IBM Web site at: http://publib.boulder.ibm.com/infocenter/db2help/
2. In the Downloads section of the welcome page under the Service and Support heading, click the **DB2 Universal Database documentation** link.
3. Determine if the version of your DB2 Information Center is out of date by comparing the latest refreshed documentation image level to the documentation level you have installed. The documentation level you have installed is listed on the DB2 Information Center welcome page.

4. If a more recent version of the DB2 Information Center is available, download the latest refreshed *DB2 Information Center* image applicable to your operating system.
5. To install the refreshed *DB2 Information Center* image, follow the instructions provided on the Web page.

Related concepts:

- "DB2 Information Center installation scenarios" on page 709

Related tasks:

- "Invoking the DB2 Information Center" on page 716
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 712
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 714

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

Procedure:

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in the Mozilla browser:

1. In Mozilla, select the **Edit** —> **Preferences** —> **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

Related concepts:

- "DB2 Information Center" on page 708

DB2 PDF and printed documentation

The following tables provide official book names, form numbers, and PDF file names. To order hardcopy books, you must know the official book name. To print a PDF file, you must know the PDF file name.

The DB2 documentation is categorized by the following headings:

- Core DB2 information
- Administration information
- Application development information
- Business intelligence information
- DB2 Connect information
- Getting started information
- Tutorial information
- Optional component information
- Release notes

The following tables describe, for each book in the DB2 library, the information needed to order the hard copy, or to print or view the PDF for that book. A full description of each of the books in the DB2 library is available from the IBM Publications Center at www.ibm.com/shop/publications/order

Core DB2 information

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

Table 95. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x81
IBM DB2 Universal Database Glossary	No form number	db2t0x81
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840, not available in hardcopy	db2m1x81
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841, not available in hardcopy	db2m2x81
IBM DB2 Universal Database What's New	SC09-4848	db2q0x81

Administration information

The information in these books covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

Table 96. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x81
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x81
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x81
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x81
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx81
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax81
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx81
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x81
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x81
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x81

Application development information

The information in these books is of special interest to application developers or programmers working with DB2 Universal Database (DB2 UDB). You will find information about supported languages and compilers, as well as the documentation required to access DB2 UDB using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLJ, and CLI. If you are using the DB2 Information Center, you can also access HTML versions of the source code for the sample programs.

Table 97. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx81
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x81
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x81

Table 97. Application development information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x81
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x81
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx81
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx81

Business intelligence information

The information in these books describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

Table 98. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Standard Edition Information Catalog Center Administration Guide	SC27-1125	db2dix81
IBM DB2 Warehouse Manager Standard Edition Installation Guide	GC27-1122	db2idx81
IBM DB2 Warehouse Manager Standard Edition Managing ETI Solution Conversion Programs with DB2 Warehouse Manager	SC18-7727	iwhe1mstx80

DB2 Connect information

The information in this category describes how to access data on mainframe and midrange servers using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

Table 99. DB2 Connect information

Name	Form number	PDF file name
IBM Connectivity Supplement	No form number	db2h1x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x81
IBM DB2 Connect User's Guide	SC09-4835	db2c0x81

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

Table 100. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832, not available in hardcopy	db2itx81
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx81
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x81
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837, not available in hardcopy	db2iyx81
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x81

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

Table 101. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux81
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax81
Information Catalog Center Tutorial	No form number	db2aix81
Video Central for e-business Tutorial	No form number	db2twx81
Visual Explain Tutorial	No form number	db2tvx81

Optional component information

The information in this category describes how to work with optional DB2 components.

Table 102. Optional component information

Name	Form number	PDF file name
IBM DB2 Cube Views Guide and Reference	SC18–7298	db2aax81

Table 102. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Query Patroller Guide: Installation, Administration and Usage Guide	GC09–7658	db2dwx81
IBM DB2 Spatial Extender and Geodetic Extender User's Guide and Reference	SC27-1226	db2sbx81
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x82
DB2 Net Search Extender Administration and User's Guide **Note:** HTML for this document is *not* installed from the HTML documentation CD.	SH12-6740	N/A

Release notes

The release notes provide additional information specific to your product's release and FixPak level. The release notes also provide summaries of the documentation updates incorporated in each release, update, and FixPak.

Table 103. Release notes

Name	Form number	PDF file name
DB2 Release Notes	See note.	See note.
DB2 Installation Notes	Available on product CD-ROM only.	Not available.

Note: The Release Notes are available in:

- XHTML and Text format, on the product CDs
- PDF format, on the PDF Documentation CD

In addition the portions of the Release Notes that discuss *Known Problems and Workarounds* and *Incompatibilities Between Releases* also appear in the DB2 Information Center.

To view the Release Notes in text format on UNIX-based platforms, see the `Release.Notes` file. This file is located in the `DB2DIR/Readme/%L` directory, where `%L` represents the locale name and `DB2DIR` represents:

- For AIX operating systems: `/usr/opt/db2_08_01`
- For all other UNIX-based operating systems: `/opt/IBM/db2/V8.1`

Related concepts:

- "DB2 documentation and help" on page 707

Related tasks:

- "Printing DB2 books from PDF files" on page 724
- "Ordering printed DB2 books" on page 724
- "Invoking contextual help from a DB2 tool" on page 725

Printing DB2 books from PDF files

You can print DB2 books from the PDF files on the *DB2 PDF Documentation* CD. Using Adobe Acrobat Reader, you can print either the entire book or a specific range of pages.

Prerequisites:

Ensure that you have Adobe Acrobat Reader installed. If you need to install Adobe Acrobat Reader, it is available from the Adobe Web site at www.adobe.com

Procedure:

To print a DB2 book from a PDF file:

1. Insert the *DB2 PDF Documentation* CD. On UNIX operating systems, mount the DB2 PDF Documentation CD. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Open index.htm. The file opens in a browser window.
3. Click on the title of the PDF you want to see. The PDF will open in Acrobat Reader.
4. Select **File → Print** to print any portions of the book that you want.

Related concepts:

- "DB2 Information Center" on page 708

Related tasks:

- "Mounting the CD-ROM (AIX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (HP-UX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (Linux)" in the *Quick Beginnings for DB2 Servers*
- "Ordering printed DB2 books" on page 724
- "Mounting the CD-ROM (Solaris Operating Environment)" in the *Quick Beginnings for DB2 Servers*

Related reference:

- "DB2 PDF and printed documentation" on page 719

Ordering printed DB2 books

If you prefer to use hardcopy books, you can order them in one of three ways.

Procedure:

Printed books can be ordered in some countries or regions. Check the IBM Publications website for your country or region to see if this service is available in your country or region. When the publications are available for ordering, you can:

- Contact your IBM authorized dealer or marketing representative. To find a local IBM representative, check the IBM Worldwide Directory of Contacts at www.ibm.com/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.

- Visit the IBM Publications Center at http://www.ibm.com/shop/publications/order. The ability to order books from the IBM Publications Center may not be available in all countries.

At the time the DB2 product becomes available, the printed books are the same as those that are available in PDF format on the *DB2 PDF Documentation CD*. Content in the printed books that appears in the *DB2 Information Center CD* is also the same. However, there is some additional content available in DB2 Information Center CD that does not appear anywhere in the PDF books (for example, SQL Administration routines and HTML samples). Not all books available on the DB2 PDF Documentation CD are available for ordering in hardcopy.

Note: The DB2 Information Center is updated more frequently than either the PDF or the hardcopy books; install documentation updates as they become available or refer to the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/ to get the most current information.

Related tasks:

- "Printing DB2 books from PDF files" on page 724

Related reference:

- "DB2 PDF and printed documentation" on page 719

Invoking contextual help from a DB2 tool

Contextual help provides information about the tasks or controls that are associated with a particular window, notebook, wizard, or advisor. Contextual help is available from DB2 administration and development tools that have graphical user interfaces. There are two types of contextual help:

- Help accessed through the **Help** button that is located on each window or notebook
- Infopops, which are pop-up information windows displayed when the mouse cursor is placed over a field or control, or when a field or control is selected in a window, notebook, wizard, or advisor and F1 is pressed.

The **Help** button gives you access to overview, prerequisite, and task information. The infopops describe the individual fields and controls.

Procedure:

To invoke contextual help:

- For window and notebook help, start one of the DB2 tools, then open any window or notebook. Click the **Help** button at the bottom right corner of the window or notebook to invoke the contextual help.

  You can also access the contextual help from the **Help** menu item at the top of each of the DB2 tools centers.

  Within wizards and advisors, click on the Task Overview link on the first page to view contextual help.
- For infopop help about individual controls on a window or notebook, click the control, then click **F1**. Pop-up information containing details about the control is displayed in a yellow window.

Note: To display infopops simply by holding the mouse cursor over a field or control, select the **Automatically display infopops** check box on the **Documentation** page of the Tool Settings notebook.

Similar to infopops, diagnosis pop-up information is another form of context-sensitive help; they contain data entry rules. Diagnosis pop-up information is displayed in a purple window that appears when data that is not valid or that is insufficient is entered. Diagnosis pop-up information can appear for:

- Compulsory fields.
- Fields whose data follows a precise format, such as a date field.

Related tasks:

- "Invoking the DB2 Information Center" on page 716
- "Invoking message help from the command line processor" on page 726
- "Invoking command help from the command line processor" on page 727
- "Invoking SQL state help from the command line processor" on page 727
- "Access to the DB2 Information Center: Concepts help"
- "How to use the DB2 UDB help: Common GUI help"
- "Setting the location for accessing the DB2 Information Center: Common GUI help"
- "Setting up access to DB2 contextual help and documentation: Common GUI help"

Invoking message help from the command line processor

Message help describes the cause of a message and describes any action you should take in response to the error.

Procedure:

To invoke message help, open the command line processor and enter:

```
? XXXnnnnn
```

where *XXXnnnnn* represents a valid message identifier.

For example, `? SQL30081` displays help about the SQL30081 message.

Related concepts:

- "Introduction to messages" in the *Message Reference Volume 1*

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking command help from the command line processor

Command help explains the syntax of commands in the command line processor.

Procedure:

To invoke command help, open the command line processor and enter:

```
? command
```

where *command* represents a keyword or the entire command.

For example, `? catalog` displays help for all of the CATALOG commands, while `? catalog database` displays help only for the CATALOG DATABASE command.

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 725
- "Invoking the DB2 Information Center" on page 716
- "Invoking message help from the command line processor" on page 726
- "Invoking SQL state help from the command line processor" on page 727

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking SQL state help from the command line processor

DB2 Univerrsal Database returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure:

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Related tasks:

- "Invoking the DB2 Information Center" on page 716
- "Invoking message help from the command line processor" on page 726
- "Invoking command help from the command line processor" on page 727

DB2 tutorials

The DB2® tutorials help you learn about various aspects of DB2 Universal Database. The tutorials provide lessons with step-by-step instructions in the areas of developing applications, tuning SQL query performance, working with data warehouses, managing metadata, and developing Web services using DB2.

Before you begin:

You can view the XHTML versions of the tutorials from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some tutorial lessons use sample data or code. See each tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

Click on a tutorial title in the following list to view that tutorial.

Business Intelligence Tutorial: Introduction to the Data Warehouse Center
Perform introductory data warehousing tasks using the Data Warehouse Center.

Business Intelligence Tutorial: Extended Lessons in Data Warehousing
Perform advanced data warehousing tasks using the Data Warehouse Center.

Information Catalog Center Tutorial
Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2® products.

DB2 documentation
Troubleshooting information can be found throughout the DB2 Information Center, as well as throughout the PDF books that make up the DB2 library. You can refer to the "Support and troubleshooting" branch of the DB2 Information Center navigation tree (in the left pane of your browser window) to see a complete listing of the DB2 troubleshooting documentation.

DB2 Technical Support Web site
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs), FixPaks and the latest listing of internal DB2 error codes, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/winos2unix/support

DB2 Problem Determination Tutorial Series
Refer to the DB2 Problem Determination Tutorial Series Web site to find information on how to quickly identify and resolve problems you might encounter while working with DB2 products. One tutorial introduces you to the DB2 problem determination facilities and tools available, and helps you decide when to use them. Other tutorials deal with related topics, such

as ″Database Engine Problem Determination″, ″Performance Problem Determination″, and ″Application Problem Determination″.

See the full set of DB2 problem determination tutorials on the DB2 Technical Support site at http://www.ibm.com/software/data/support/pdm/db2tutorials.html

Related concepts:

- “DB2 Information Center” on page 708
- “Introduction to problem determination - DB2 Technical Support tutorial” in the *Troubleshooting Guide*

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. The following list specifies the major accessibility features in DB2® Version 8 products:

- All DB2 functionality is available using the keyboard for navigation instead of the mouse. For more information, see “Keyboard input and navigation.”
- You can customize the size and color of the fonts on DB2 user interfaces. For more information, see “Accessible display.”
- DB2 products support accessibility applications that use the Java™ Accessibility API. For more information, see “Compatibility with assistive technologies” on page 730.
- DB2 documentation is provided in an accessible format. For more information, see “Accessible documentation” on page 730.

Keyboard input and navigation

Keyboard input

You can operate the DB2 tools using only the keyboard. You can use keys or key combinations to perform operations that can also be done using a mouse. Standard operating system keystrokes are used for standard operating system operations.

For more information about using keys or key combinations to perform operations, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard navigation

You can navigate the DB2 tools user interface using keys or key combinations.

For more information about using keys or key combinations to navigate the DB2 Tools, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard focus

In UNIX® operating systems, the area of the active window where your keystrokes will have an effect is highlighted.

Accessible display

The DB2 tools have features that improve accessibility for users with low vision or other visual impairments. These accessibility enhancements include support for customizable font properties.

Font settings

You can select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

For more information about specifying font settings, see Changing the fonts for menus and text: Common GUI help.

Non-dependence on color

You do not need to distinguish between colors in order to use any of the functions in this product.

Compatibility with assistive technologies

The DB2 tools interfaces support the Java Accessibility API, which enables you to use screen readers and other assistive technologies with DB2 products.

Accessible documentation

Documentation for DB2 is provided in XHTML 1.0 format, which is viewable in most Web browsers. XHTML allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

Syntax diagrams are provided in dotted decimal format. This format is available only if you are accessing the online documentation using a screen-reader.

Related concepts:

- "Dotted decimal syntax diagrams" on page 730

Related tasks:

- "Keyboard shortcuts and accelerators: Common GUI help"
- "Changing the fonts for menus and text: Common GUI help"

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.
- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one

data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Related concepts:

- "Accessibility" on page 729

Related tasks:

- "Keyboard shortcuts and accelerators: Common GUI help"

Related reference:

- "How to read the syntax diagrams" in the *SQL Reference, Volume 2*

Common Criteria certification of DB2 Universal Database products

DB2 Universal Database is being evaluated for certification under the Common Criteria at evaluation assurance level 4 (EAL4). For more information about Common Criteria, see the Common Criteria web site at: http://niap.nist.gov/cc-scheme/.

Appendix F. Notices

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

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

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 of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

ACF/VTAM
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
BookManager
C Set++
C/370
CICS
Database 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Information Integrator
DB2 Query Patroller
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eServer
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
iSeries
LAN Distance
MVS
MVS/ESA
MVS/XA
Net.Data
NetView
OS/390
OS/400
PowerPC
pSeries
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/400
SQL/DS
System/370
System/390
SystemView
Tivoli
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WebSphere
WIN-OS/2
z/OS
zSeries

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 UDB documentation library:

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

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-888-426-4343 to learn about available service options
- 1-800-IBM-4YOU (426-4968) for DB2 marketing and sales

In Canada, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-800-465-9600 to learn about available service options
- 1-800-IBM-4YOU (1-800-426-4968) for DB2 marketing and sales

To locate an IBM office in your country or region, check IBM's Directory of Worldwide Contacts on the web at http://www.ibm.com/planetwide

Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at http://www.ibm.com/software/data/db2/udb

This site contains the latest information on the technical library, ordering books, product downloads, newsgroups, FixPaks, news, and links to web resources.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-IBM-CALL (1-800-426-2255) to order products or to obtain general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, go to the IBM Worldwide page at www.ibm.com/planetwide

Printed in USA

SC09-4826-01

Spine information:

IBM® DB2 Universal Database™ Programming Client Applications Version 8.2