IBM® DB2 Universal Database™

Application Development Guide: Programming Client Applications

Version 8

SC09-4826-00

IBM® DB2 Universal Database™

Application Development Guide: Programming Client Applications

Version 8

SC09-4826-00

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.
  - Java (both JDBC and SQLj), and considerations for building Java applications that use WebSphere Application Servers.
  - The IBM OLE DB Provider for DB2 Servers. General information is provided about IBM OLE DB Provider support for OLE DB services, components, and properties. Specific information is also provided about Visual Basic and Visual C++ applications that use the OLE DB interface for ActiveX Data Objects (ADO).
  - National language support issues. General topics, such as collating sequences, the derivation of code pages and locales, and character conversions are described. More specific issues such as DBCS code pages, EUC character sets, and issues that apply in Japanese and Traditional Chinese EUC and UCS-2 environments are also described.
  - Transaction management. Issues that apply to applications that perform multisite updates, and to applications that perform concurrent transactions, are described.
  - Applications in partitioned database environments. Directed DSS, local bypass, buffered inserts, and troubleshooting applications in partitioned database environments are described.

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

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

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

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

Part 1. Introduction

Chapter 1. Overview of Supported Programming Interfaces

DB2 Developer's Edition

The sections that follow describe the DB2 Developer's Edition, and where to find information about installing products in it.

DB2 Developer's Edition Products

DB2® Universal Database provides two product packages for application development: DB2 Personal Developer's Edition and DB2 Universal Developer's Edition. The Personal Developer's Edition provides the DB2 Universal Database™ and DB2 Connect™ Personal Edition products that run on Linux and Windows® operating systems. The DB2 Universal Developer's Edition provides DB2 products on these platforms as well as on AIX, HP-UX, and the Solaris Operating Environment. Contact your IBM representative for a full list of supported platforms.

Using the software that comes with these products, you can develop and test applications that run on one operating system and access databases on the same or on a different operating system. For example, you can create an application that runs on the Windows NT® operating system but accesses a database on a UNIX® platform such as AIX. See your License Agreement for the terms and conditions of use for the Developer's Edition products.

The Personal Developer's Edition contains several CD-ROMs with all the code that you need to develop and test your applications. In each box, you will find:

- The DB2 Universal Database product CD-ROMs for Linux and Windows operating systems. Each CD-ROM contains the DB2 server, Administration Client, Application Development Client, and Run-Time Client for a supported operating system. These CD-ROMs are provided to you for testing your applications only. If you need to install and use a database, you have to get a valid license by purchasing the Universal Database product.
- DB2 Connect Personal Edition
- A DB2 publications CD-ROM containing DB2 books in PDF format
- DB2 Extenders™ (Windows only)
- DB2 XML Extender (Windows only)
- VisualAge® for Java, Entry Edition

The Universal Developer's Edition contains CD-ROMs for all the operating systems supported by DB2, and include the following:

- DB2 Universal Database Personal Edition, Workgroup Server Edition, and Enterprise Server Edition
- DB2 Connect Personal Edition and DB2 Connect Enterprise Edition
- Administration clients for all platforms. These clients contain tools for administering databases, such as the Control Center and the Event Analyzer. These clients also allow you to run applications on any system.
- Application development clients for all platforms. These clients have application development tools, sample programs, and header files. Each DB2 AD client includes everything you need to develop your applications.
- Run-time clients for all platforms. An application can be run from a run-time client on any system. The run-time client does not have some of the features of the administration client, such as the DB2 Control Center and Event Analyzer, and so takes up less space.
- DB2 Extenders
- DB2 XML Extender
- VisualAge for Java, Professional Edition (Windows)
- Websphere Studio
- Websphere Application Server, Standard Edition
- Query Management Facility (try and buy)

In addition, for both Developer's Editions you get copies of other software that you may find useful for developing applications. This software may vary from time to time, and is accompanied by license agreements for use.

Instructions for Installing DB2 Developer's Edition Products

For instructions on how to install a product that is available with DB2 Developer's Edition, either refer to the appropriate *Quick Beginnings* book, which is available from the PDF CD, or check the product CD itself for installation instructions.

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 Center
: 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.

Script 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.

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.

Alert Center
Is used to monitor your system for early warnings of potential problems, or to automate actions to correct problems.

Tools Setting
Is used to change the settings for the Control Center, Alert Center, and Replication.

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.

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 Object Linking and Embedding Database (OLE DB) Bridge.
- 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 155
- "Embedded SQL" on page 9
- "DB2 Call Level Interface" on page 10
- "DB2 Application Programming Interfaces" on page 8
- "ActiveX Data Objects and Remote Data Objects" on page 16
- "Perl DBI" on page 17
- "ODBC End-User Tools" on page 17
- "Tools for Building Web Applications" on page 17
- "Java Database Connectivity (JDBC)" on page 14

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 58
- "Administrative APIs in Embedded SQL or DB2 CLI Programs" on page 48

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 336
- "Embedded SQL Statements in C and C++" on page 167
- "Embedded SQL Statements in COBOL" on page 217
- "Embedded SQL Statements in FORTRAN" on page 242
- "Embedded SQL Statements in Java" on page 278
- "Embedded SQL for Java (SQLj)" on page 15

Related tasks:

- "Embedding SQL Statements in a Host Language" on page 71

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. For more information, see:

`http://www.ibm.com/software/data/db2/udb/staticcli`

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, 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 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 48
- "DB2 Call Level Interface (CLI) versus Embedded Dynamic SQL" on page 155
- "Advantages of DB2 CLI over Embedded SQL" on page 157

- "When to Use DB2 CLI or Embedded SQL" on page 159
- "DB2 CLI versus Embedded Dynamic SQL" on page 12

Related reference:

- "DB2 CLI Samples" in the *Application Development Guide: Building and Running Applications*

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 155
- "Advantages of DB2 CLI over Embedded SQL" on page 157
- "When to Use DB2 CLI or Embedded SQL" on page 159

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/java`

Related concepts:

• "Comparison of SQLj to JDBC" on page 258

Related tasks:

• "Coding JDBC Applications and Applets" on page 268

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/java`

Related concepts:

- "Comparison of SQLj to JDBC" on page 258

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 329

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.

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 18
- "XML Extender" on page 19

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 317

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`

Related concepts:

- "Document Access Definition Extension File" on page 312

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 17
- "XML Extender" on page 19

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 22
- "DB2 User-Defined Functions and Methods" on page 22

- "User-Defined Types (UDTs) and Large Objects (LOBs)" on page 24
- "DB2 Triggers" on page 27

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 23

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 22
- "OLE Automation Routines" on page 26

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 22
- "Development Center" on page 23

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 355
- "OLE DB Services Automatically Enabled by IBM OLE DB Provider" on page 359

Related reference:

- "IBM OLE DB Provider Support for OLE DB Components and Interfaces" on page 366
- "IBM OLE DB Provider Support for OLE DB Properties" on page 369

DB2 Triggers

A trigger defines a set of actions executed by a delete, insert, or update 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 30
- "Application Pseudocode Framework" on page 45
- "Facilities for Prototyping SQL Statements" on page 46
- "Sample Files" in the *Application Development Guide: Building and Running Applications*
- "Sample Programs: Structure and Design" in the *Application Development Guide: Building and Running Applications*

Related tasks:

- "Creating the Declaration Section of a Standalone Application" on page 31
- "Connecting an Application to a Database" on page 40
- "Coding Transactions" on page 41
- "Ending a Transaction with the COMMIT Statement" on page 42
- "Ending a Transaction with the ROLLBACK Statement" on page 43
- "Ending an Application Program" on page 44
- "Setting up a Testing Environment" on page 59

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 concepts:

- "SQLSTATE and SQLCODE Values in Java" on page 304

Related tasks:

- "Declaring Variables That Interact with the Database Manager" on page 32
- "Declaring Variables That Represent SQL Objects" on page 33
- "Relating Host Variables to an SQL Statement" on page 36
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 35
- "Declaring the SQLCA for Error Handling" on page 37

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 200
- "Supported SQL Data Types in COBOL" on page 231
- "Supported SQL Data Types in FORTRAN" on page 251
- "Supported SQL Data Types in Java" on page 264
- "Supported SQL Data Types in REXX" on page 345

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 169
- "Syntax for Fixed and Null-Terminated Character Host Variables in C and C++" on page 173
- "C Macro Expansion" on page 184
- "Host Variables in COBOL" on page 219
- "Host Variables in FORTRAN" on page 244
- "Host Variables in Java" on page 263
- "Host Variables in REXX" on page 338

Related reference:

- "Syntax for Numeric Host Variables in C and C++" on page 172
- "Syntax for Variable-Length Character Host Variables in C or C++" on page 174
- "Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++" on page 177
- "Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++" on page 178
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 179
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 182
- "Syntax for File Reference Host Variable Declarations in C or C++" on page 183

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[5];
```

```
  short years;
  double salary;
  double comm;
} staff;
```

Related reference:

- "db2dclgn - Declaration Generator" 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 32
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 35
- "Creating the Declaration Section of a Standalone Application" on page 31

Related reference:

- "Supported SQL Data Types in C and C++" on page 200
- "Supported SQL Data Types in COBOL" on page 231
- "Supported SQL Data Types in FORTRAN" on page 251
- "Supported SQL Data Types in Java" on page 264
- "Supported SQL Data Types in REXX" on page 345

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 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:

- "Definition of FIPS 127-2 and ISO/ANS SQL92" on page 48
- "Error Handling Using the WHENEVER Statement" on page 38
- "SQLSTATE and SQLCODE Variables in C and C++" on page 206
- "SQLSTATE and SQLCODE Variables in COBOL" on page 235
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 253
- "SQLSTATE and SQLCODE Values in Java" on page 304
- "SQLSTATE and SQLCODE Variables in Perl" on page 331

Related tasks:

- "Creating the Declaration Section of a Standalone Application" on page 31

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 31

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 419

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 42
- "Ending a Transaction with the ROLLBACK Statement" on page 43

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 45
- "Return Codes" on page 123

- "Error Information in the SQLCODE, SQLSTATE, and SQLWARN Fields" on page 123

Related tasks:

- "Ending an Application Program" on page 44

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 45
- "Remote Unit of Work" on page 419
- "Multisite Update" on page 419

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 45

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 207

Related tasks:

- "Ending an Application Program" on page 44

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

Related tasks:

- "Programming a Standalone Application" on page 30

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 5

Related reference:

- Appendix A, "Supported SQL Statements" on page 475

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 58
- "Sample Programs: Structure and Design" in the *Application Development Guide: Building and Running Applications*

Definition of FIPS 127-2 and ISO/ANS SQL92

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*

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 49
- "Data Value Control Using Unique Constraints" on page 49
- "Data Value Control Using Table Check Constraints" on page 50
- "Data Value Control Using Referential Integrity Constraints" on page 50
- "Data Value Control Using Views with Check Option" on page 51
- "Data Value Control Using Application Logic and Program Variable Types" on page 51

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 52
- "Referential Integrity Differences among IBM Relational Database Systems" on page 488

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 51

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" on page 52
- "Data Relationship Control Using Triggers" on page 52

- "Data Relationship Control Using Before Triggers" on page 53
- "Data Relationship Control Using After Triggers" on page 53
- "Data Relationship Control Using Application Logic" on page 54

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 53
- "Data Relationship Control Using After Triggers" on page 53
- "DB2 Triggers" on page 27

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 53
- "DB2 Triggers" on page 27

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 53
- "DB2 Triggers" on page 27

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 54

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 53
- "Data Relationship Control Using After Triggers" on page 53
- "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 22
- "DB2 User-Defined Functions and Methods" on page 22
- "DB2 Triggers" on page 27

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 57
- "Authorization Considerations for Static SQL" on page 58
- "Authorization Considerations for APIs" on page 58

- "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 55
- "Authorization Considerations for Static SQL" on page 58
- "Authorization Considerations for APIs" on page 58
- "Effects of DYNAMICRULES on Dynamic SQL" on page 135

Related reference:

- "BIND" 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 55
- "Authorization Considerations for Dynamic SQL" on page 57
- "Authorization Considerations for APIs" on page 58

Related reference:

- "BIND" 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 55
- "Authorization Considerations for Dynamic SQL" on page 57
- "Authorization Considerations for Static SQL" on page 58

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 60
- "Generating Test Data" on page 61
- "Debugging and Optimizing an Application" on page 63

Related reference:

- "sqlecrea - Create Database" in the *Administrative API Reference*

- "CREATE DATABASE" 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 61

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 22

Related tasks:

- "Debugging and Optimizing an Application" on page 63

Related reference:

- "INSERT scalar function" in the *SQL Reference, Volume 1*
- "RESTORE DATABASE" 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 OS/390 and z/OS, 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 46
- "The database system-monitor information" in the *Administration Guide: Performance*
- "Source File Requirements for Embedded SQL Applications" on page 80

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

The sections that follow describe the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++.

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

The IBM® DB2® Universal Database Project Add-In for Microsoft® Visual C++ is a collection of management tools and wizards that plug into the Visual C++ component of Visual Studio IDE. The tools and wizards automate and simplify the various tasks involved in developing applications for DB2 using embedded SQL.

You can use the IBM DB2 Universal Database™ Project Add-In for Microsoft Visual C++ to develop, package, and deploy:

- Stored procedures written in C/C++ for DB2 Universal Database on Windows® operating systems
- Windows C/C++ embedded SQL client applications that access DB2 Universal Database servers
- Windows C/C++ client applications that invoke stored procedures using C/C++ function call wrappers

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ allows you to focus on the design and logic of your DB2 applications rather than the actual building and deployment of it.

Some of the tasks performed by the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ include:

- Creating a new embedded SQL module
- Inserting SQL statements into an embedded SQL module using SQL Assist
- Adding imported stored procedures
- Creating an exported stored procedure
- Packaging the DB2 Project
- Deploying the DB2 project from within Visual C++

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ is presented in the form of a toolbar. The toolbar buttons include:

DB2 Project Properties
: Manages the project properties (development database and code-generation options)

New DB2 Object
: Adds a new embedded SQL module, imported stored procedure, or exported stored procedure

DB2 Embedded SQL Modules
Manages the list of embedded SQL modules and their precompiler options

DB2 Imported Stored Procedures
Manages the list of imported stored procedures

DB2 Exported Stored Procedures
Manages the list of exported stored procedures

Package DB2 Project
Packages the DB2 external project files

Deploy DB2 Project
Deploys the packaged DB2 external project files

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ also has the following three hidden buttons that can be made visible using the standard Visual C++ tools customization options:

New DB2 Embedded SQL Module
Adds a new C/C++ embedded SQL module

New DB2 Imported Stored Procedure
Imports a new database stored procedure

New DB2 Exported Stored Procedure
Exports a new database stored procedure

The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ can automatically generate the following code elements:

- Skeletal embedded SQL module files with optional sample SQL statements
- Standard database connect and disconnect embedded SQL functions
- Imported stored procedure call wrapper functions
- Exported stored procedure function templates
- Exported stored procedure data definition language (DDL) files

For more information on the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++, refer to:

- The online help for the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++
- http://www.ibm.com/software/data/db2/udb/ide/index.html

Related tasks:

- "Activating the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 67

- "Activating the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++" on page 68

Related reference:

- "IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ Terminology" on page 66

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ Terminology

The terminology associated with the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ is as follows:

IDE project
The standard Visual C++ project

DB2 project
The collection of DB2 project objects that are inserted into the IDE project. DB2 project objects can be inserted into any Visual C++ project. The DB2 project allows you to manage the various DB2 objects such as embedded SQL modules, imported stored procedures, and exported stored procedures. You can add, delete, and modify these objects and their properties.

module
A C/C++ source code file that might contain SQL statements.

development database
The database that is used to compile embedded SQL modules. The development database is also used to look up the list of importable database stored procedure definitions.

embedded SQL module
A C/C++ source code file that contains embedded static or dynamic SQL.

imported stored procedure
A stored procedure, already defined in the database, that the project invokes.

exported stored procedure
A database stored procedure that is built and defined by the project.

Related concepts:

- "The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 64

Related tasks:

- "Activating the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 67

- "Activating the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++" on page 68

Activating the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

Activate the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ to access the floating toolbar.

Note: If the toolbar is accidentally closed, you can either deactivate then reactivate the add-in or use the Microsoft Visual C++ standard customization options to redisplay the toolbar.

Procedure:

1. Start and stop Visual C++ at least once with your current login ID. The first time you run Visual C++, a profile is created for your user ID, and that is what gets updated by the `db2vccmd` command. If you have not started it once, and you try to run `db2vccmd`, you may see errors like the following:

   ```
   "Registering DB2 Project add-in ...Failed! (rc = 2)"
   ```

2. Register the add-in, if you have not already done so, by entering the following on the command line:

   ```
   db2vccmd register
   ```

3. Select **Tools —> Customize**. The Customize notebook opens.
4. Select the **Add-ins and Macro Files** tab. The Add-ins and Macro Files page opens.
5. Select the **IBM DB2 Project Add-In** check box.
6. Click **OK**. A floating toolbar will be created.

Related concepts:

- "The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 64

Related tasks:

- "Activating the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++" on page 68

Related reference:

- "IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ Terminology" on page 66

Activating the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++

The DB2 Tools Add-In is a toolbar that enables the launch of some of the DB2 administration and development tools from within the Visual C++ integrated development environment.

Procedure:

To activate the IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++, perform the following steps:

1. Start and stop Visual C++ at least once with your current login ID. The first time you run Visual C++, a profile is created for your user ID, and that is what gets updated by the `db2vccmd` command. If you have not started it once, and you try to run `db2vccmd`, you may see errors like the following:

   ```
   "Registering DB2 Project add-in ...Failed! (rc = 2)"
   ```

2. Register the add-in, if you have not already done so, by entering the following on the command line:

   ```
   db2vccmd register
   ```

3. Select **Tools —> Customize**. The Customize notebook opens.
4. Select the Add-ins and Macro Files tab.
5. Select the **IBM DB2 Tools Add-In** check box.
6. Click **OK**. A floating toolbar will be created.

Note: If the toolbar is accidentally closed, you can either deactivate then reactivate the add-in or use the Visual C++ standard customization options to redisplay the toolbar.

Related concepts:

- "The IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 64

Related tasks:

- "Activating the IBM DB2 Universal Database Project Add-In for Microsoft Visual C++" on page 67

Related reference:

- "IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ Terminology" on page 66

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.`

Table 2. Embedding SQL Statements in a Host Language (continued)

Language	Sample Source Code

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 */`

Table 3. Embedding SQL Statements in a Host Language (continued)

Language	Sample Source Code

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 336
- "Embedded SQL Statements in C and C++" on page 167
- "Embedded SQL Statements in COBOL" on page 217
- "Embedded SQL Statements in FORTRAN" on page 242
- "Embedded SQL Statements in Java" on page 278

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.

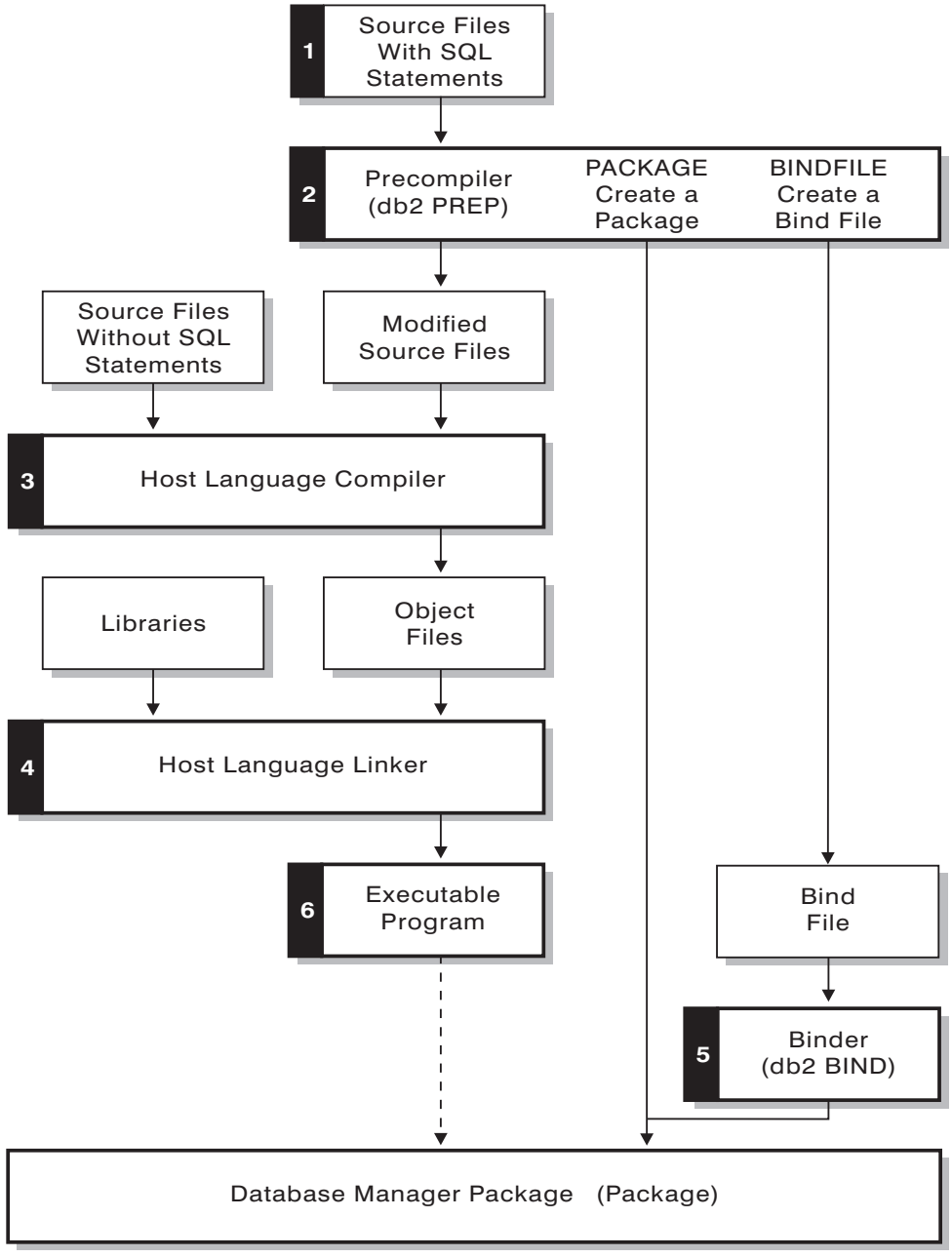


Figure 1. Preparing Programs Written in Compiled Host Languages

Related concepts:

- "Precompilation of Source Files Containing Embedded SQL" on page 78

- "Source File Requirements for Embedded SQL Applications" on page 80
- "Compilation and Linkage of Source Files Containing Embedded SQL" on page 81
- "Embedded SQL" on page 9

Related reference:

- "BIND" 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:

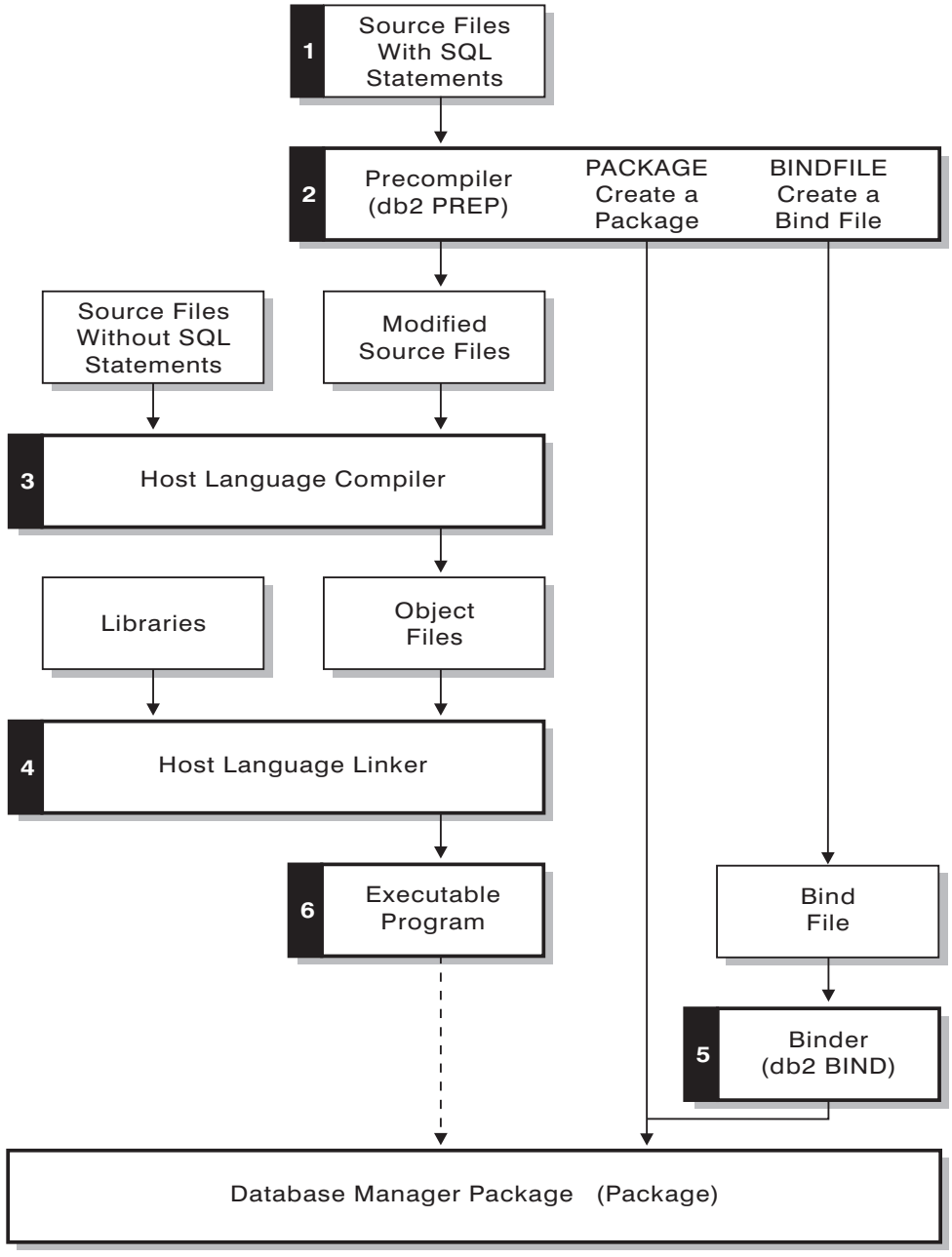


Figure 2. Preparing Programs Written in Compiled Host Languages

- 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 78
- "Source File Requirements for Embedded SQL Applications" on page 80
- "Compilation and Linkage of Source Files Containing Embedded SQL" on page 81
- "Package Creation Using the BIND Command" on page 83
- "Package Versioning" on page 83
- "Effect of Special Registers on Bound Dynamic SQL" on page 85
- "Resolution of Unqualified Table Names" on page 85
- "Additional Considerations when Binding" on page 86
- "Advantages of Deferred Binding" on page 87
- "Application, Bind File, and Package Relationships" on page 88
- "Precompiler-Generated Timestamps" on page 88
- "Package Rebinding" on page 90
- "SQLj Translator Options" on page 284

Related reference:

- "db2bfd - Bind File Description Tool" 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 83

Related reference:

- "PRECOMPILE" 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 OS/390 and z/OS, 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 OS/390 and z/OS. 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 OS/390 and z/OS 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 87
- "Character Conversion Between Different Code Pages" on page 397
- "When Code Page Conversion Occurs" on page 397
- "Character Substitutions During Code Page Conversions" on page 398
- "Supported Code Page Conversions" on page 399
- "Code Page Conversion Expansion Factor" on page 400

Related reference:

- "PRECOMPILE" 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 22

Related tasks:

- "Building and Running REXX Applications" on page 347
- "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 C Applications on AIX" in the *Application Development Guide: Building and Running Applications*
- "Building C++ Applications on AIX" 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 Micro Focus COBOL Applications on AIX" 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" in the *Command Reference*
- "PRECOMPILE" 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 88

Related reference:

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

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" 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 397
- "Character Substitutions During Code Page Conversions" on page 398
- "Code Page Conversion Expansion Factor" on page 400

Related reference:

- "BIND" 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" 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 129

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 83

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" in the *Command Reference*
- "REBIND" 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 97
- "Error Message Retrieval in an Application" on page 126

Related tasks:

- "Declaring Host Variables in Static SQL Programs" on page 99
- "Selecting Multiple Rows Using a Cursor" on page 108
- "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 (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 97
- "Example of a Cursor in a Static SQL Program" on page 112

Related tasks:

- "Declaring Host Variables in Static SQL Programs" on page 99
- "Referencing Host Variables in Static SQL Programs" on page 101
- "Including Indicator Variables in Static SQL Programs" on page 101
- "Selecting Multiple Rows Using a Cursor" on page 108
- "Declaring and Using Cursors in Static SQL Programs" on page 109

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 169
- "Host Variables in COBOL" on page 219
- "Host Variables in FORTRAN" on page 244
- "Host Variables in Java" on page 263
- "Host Variables in REXX" on page 338

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 35
- "Declaring Host Variables in Static SQL Programs" on page 99
- "Referencing Host Variables in Static SQL Programs" on page 101

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 appplication.

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 35
- "Referencing Host Variables in Static SQL Programs" on page 101

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 35
- "Declaring Host Variables in Static SQL Programs" on page 99

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" );`

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 35
- "Declaring Host Variables in Static SQL Programs" on page 99
- "Referencing Host Variables in Static SQL Programs" on page 101

Related reference:

- "Data Types for Indicator Variables in Static SQL Programs" on page 104

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 397

Related reference:

- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "Supported SQL Data Types in C and C++" on page 200
- "Supported SQL Data Types in COBOL" on page 231
- "Supported SQL Data Types in FORTRAN" on page 251
- "Supported SQL Data Types in Java" on page 264
- "Supported SQL Data Types in REXX" on page 345

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 95

Related tasks:

- "Including Indicator Variables in Static SQL Programs" on page 101

Related reference:

- "Data Types for Indicator Variables in Static SQL Programs" on page 104

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 112

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 110

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 108

Related reference:

- "Cursor Types" on page 114

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 108

• "Declaring and Using Cursors in Static SQL Programs" on page 109

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:

```
 /* delcare 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 defintion
  #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 110
- "Error Message Retrieval in an Application" on page 126

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 108
- "Declaring and Using Cursors in Static SQL Programs" on page 109

Related reference:

- "Cursor Types" on page 114

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 (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 concepts:

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

Related reference:

- "PRECOMPILE" in the *Command Reference*

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 360

Related tasks:

- "Updating and Deleting Retrieved Data in Static SQL Programs" on page 114

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 126

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 (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 concepts:

- "JDBC Specification" on page 268

Related tasks:

- "Keeping a Copy of the Data" on page 117
- "Retrieving Data a Second Time" on page 118

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 119

Related tasks:

- "Retrieving Data a Second Time" on page 118

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 119

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 118

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 114
- "Scrolling Through Previously Retrieved Data" on page 117

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 126

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 (SQLJ)"
- "TbMod.sqlj -- How to modify table data (SQLj)"
- "TutMod.out -- HOW TO MODIFY TABLE DATA (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 SQLSTATEs when you are concerned about portability since SQLSTATEs 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:

- "SQLSTATE and SQLCODE Variables in C and C++" on page 206
- "SQLSTATE and SQLCODE Variables in COBOL" on page 235
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 253
- "SQLSTATE and SQLCODE Values in Java" on page 304
- "SQLSTATE and SQLCODE Variables in Perl" on page 331

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 485

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 206
- "SQLSTATE and SQLCODE Variables in COBOL" on page 235
- "SQLSTATE and SQLCODE Variables in FORTRAN" on page 253
- "SQLSTATE and SQLCODE Values in Java" on page 304
- "SQLSTATE and SQLCODE Variables in Perl" on page 331

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 128
- "Dynamic SQL Versus Static SQL" on page 129

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 475

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

Table 8. Comparing Static and Dynamic SQL (continued)

Consideration	Likely Best Choice
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 154

Related tasks:

- "Providing Variable Input to Dynamic SQL Using Parameter Markers" on page 153

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 133

- "Cursors in REXX" on page 344

Related tasks:

- "Selecting Multiple Rows Using a Cursor" on page 108

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 (execept 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 126

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 DYNAMICRULES on Dynamic SQL

The PRECOMPILE and BIND option DYNAMICRULES determines what values apply at run-time for the following dynamic SQL attributes:

- The authorization ID that is used during authorization checking.
- The qualifier that is used for qualification of unqualified objects.
- Whether the package can be used to dynamically prepare the following statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE statements.

In addition to the DYNAMICRULES value, the run-time environment of a package controls how dynamic SQL statements behave at run-time. The two possible run-time environments are:

- The package runs as part of a stand-alone program
- The package runs within a routine context

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

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

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

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

Invoke behavior

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

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

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

Table 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
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 57

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 133

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 SQLVARs, and secondary SQLVARs.

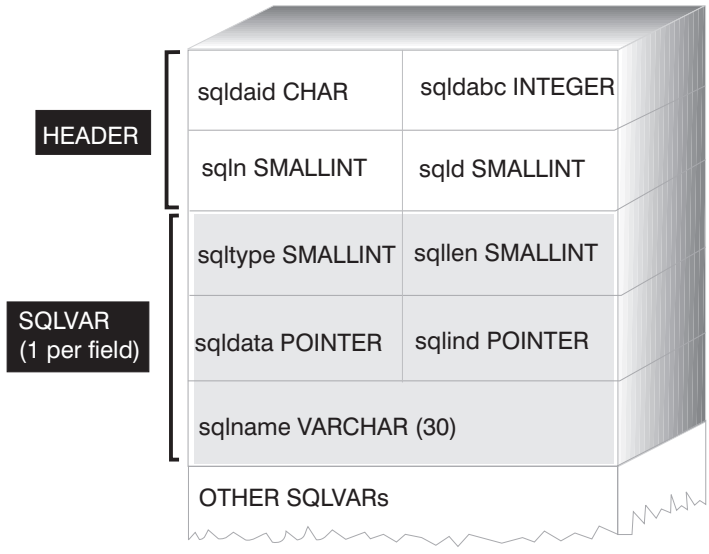


Figure 3. The SQL Descriptor Area (SQLDA)

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 SQLVARs 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 140
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 142
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 145

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 SQLVARs 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 SQLVARs 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 SQLVARs 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 SQLVARs 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 SQLVARs 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 SQLCODEs 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 SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs 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 138
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 142
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 145

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 138
- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 140
- "Allocating an SQLDA Structure for a Dynamic SQL Program" on page 145

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 144

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 145

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 SQLVARs and hence the amount of storage that the SQLDA uses. For example, to change the default number of SQLVARs 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;`

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_sqlname1
      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_sqlname1                                    )
      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 140
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 142
- "Transferring Data in a Dynamic SQL Program Using an SQLDA Structure" on page 149

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

Table 12. DB2 SQLDA SQL Types (continued). Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name[1]
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

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 143
- "Acquiring Storage to Hold a Row" on page 144
- "Processing the Cursor in a Dynamic SQL Program" on page 145

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 151

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 138
- "Preparing a Statement in Dynamic SQL Using the Minimum SQLDA Structure" on page 140
- "Allocating an SQLDA with Sufficient SQLVAR Entries for a Dynamic SQL Program" on page 142
- "Describing a SELECT Statement in a Dynamic SQL Program" on page 143
- "Acquiring Storage to Hold a Row" on page 144
- "Processing the Cursor in a Dynamic SQL Program" on page 145

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 SQLVARs 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 fuction `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 126

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 (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 recommeded you use the `SQLEndTran()` function to specify either the COMMIT or ROLLBACK statement.

Related reference:

- Appendix A, "Supported SQL Statements" on page 475

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 OS/390 and z/OS 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 updateable 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 285

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/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.

SQLEAU (sqleau.h)
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.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 166

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 184

Related reference:

- "PREPARE statement" in the *SQL Reference, Volume 2*
- "Include Files for C and C++" on page 163

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 475

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 170
- "Host Variable Declarations in C and C++" on page 171
- "Syntax for Fixed and Null-Terminated Character Host Variables in C and C++" on page 173
- "Indicator Variables in C and C++" on page 176
- "Graphic Host Variables in C and C++" on page 176
- "Host Variable Initialization in C and C++" on page 183
- "Host Structure Support in C and C++" on page 185
- "SQL Declare Section with Host Variables for C and C++" on page 198

Related reference:

- "Syntax for Numeric Host Variables in C and C++" on page 172
- "Syntax for Variable-Length Character Host Variables in C or C++" on page 174
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 179
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 182

- "Syntax for File Reference Host Variable Declarations in C or C++" on page 183

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 171

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 173
- "Graphic Host Variables in C and C++" on page 176
- "Host Structure Support in C and C++" on page 185
- "Class Data Members Used as Host Variables in C and C++" on page 191

Related tasks:

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 35
- "Declaring Structured Type Host Variables" in the *Application Development Guide: Programming Server Applications*

Related reference:

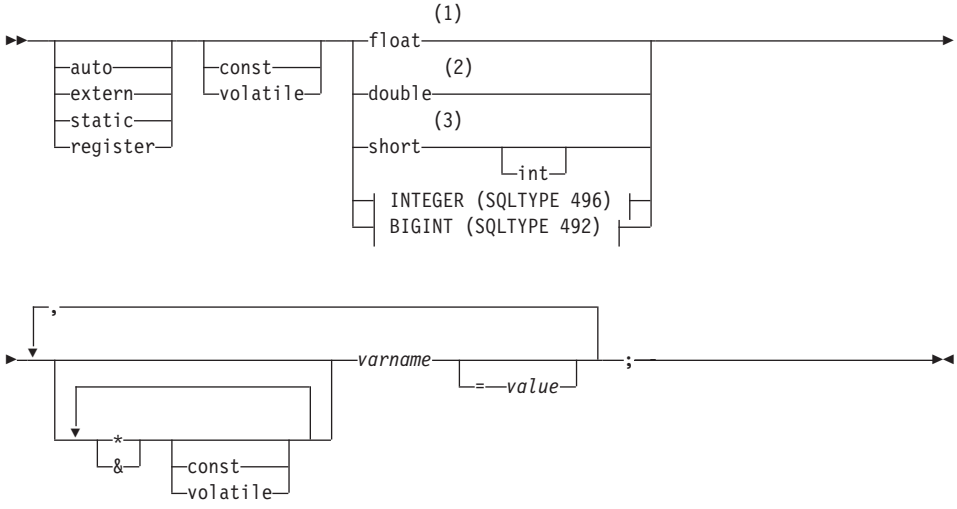
- "Syntax for Numeric Host Variables in C and C++" on page 172

- "Syntax for Variable-Length Character Host Variables in C or C++" on page 174

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)

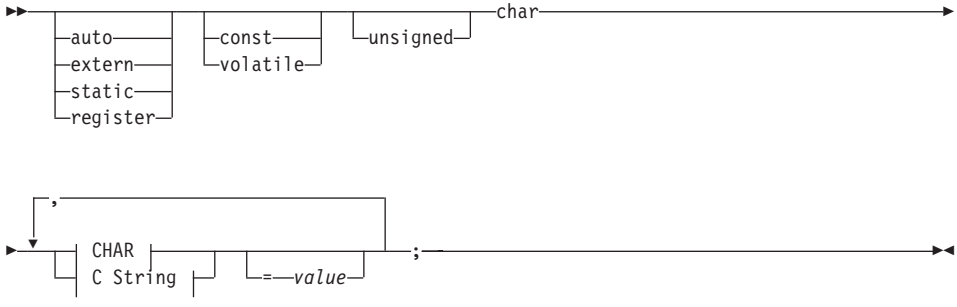
4 For maximum application portability, use sqlint32 and sqlint64 for INTEGER and BIGINT host variables, respectively. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO 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.

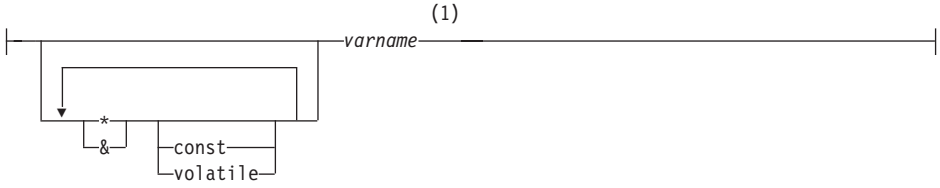
Syntax for Fixed and Null-Terminated Character Host Variables in C and C++

Following is the syntax for declaring fixed and null-terminated character host variables in C or C++.

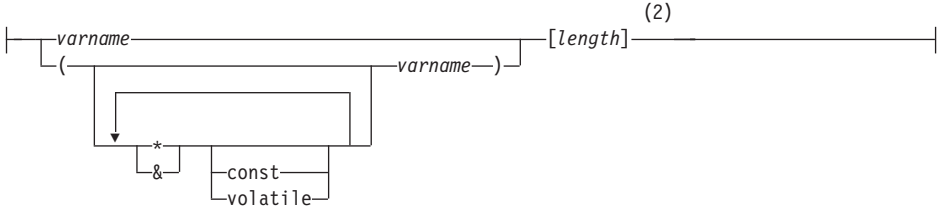
Syntax for Fixed and Null-Terminated Character Host Variables in C or C++



CHAR



C String



Notes:

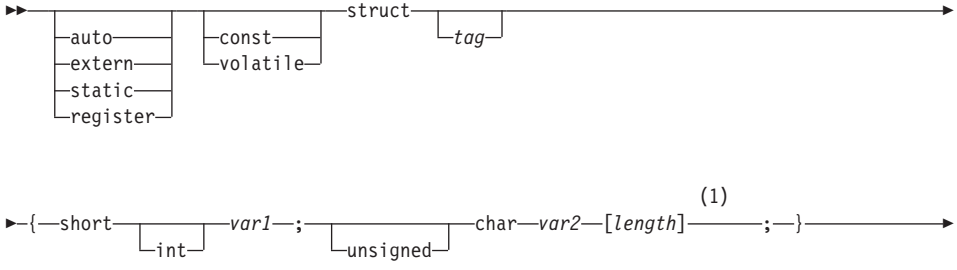
1 CHAR (SQLTYPE 452), length 1

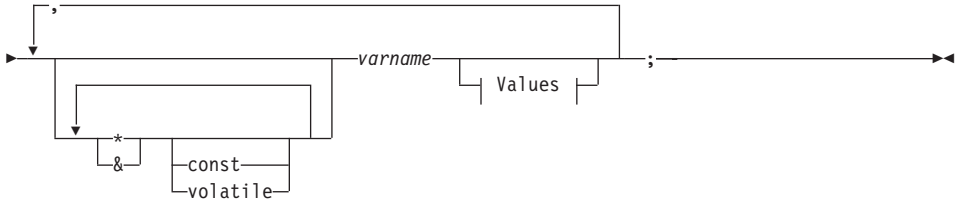
2 Null-terminated C string (SQLTYPE 460); length can be any valid constant expression

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 or 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 190

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 187

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 170
- "Host Variable Declarations in C and C++" on page 171
- "Host Variable Initialization in C and C++" on page 183
- "Host Structure Support in C and C++" on page 185
- "Indicator Tables in C and C++" on page 187
- "Multi-Byte Character Encoding in C and C++" on page 192
- "wchar_t and sqldbchar Data Types in C and C++" on page 193
- "WCHARTYPE Precompiler Option in C and C++" on page 194

Related reference:

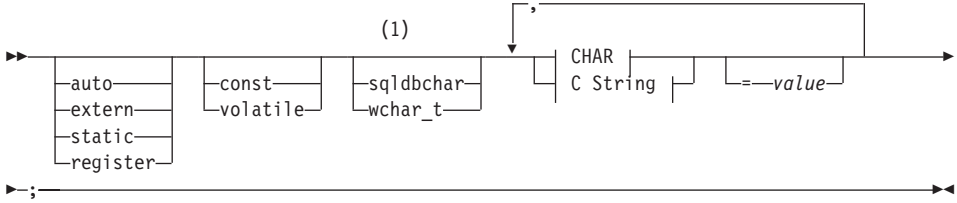
- "Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++" on page 177

- "Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++" on page 178
- "Syntax for Large Object (LOB) Host Variables in C or C++" on page 179
- "Syntax for Large Object (LOB) Locator Host Variables in C or C++" on page 182
- "Syntax for File Reference Host Variable Declarations in C or C++" on page 183

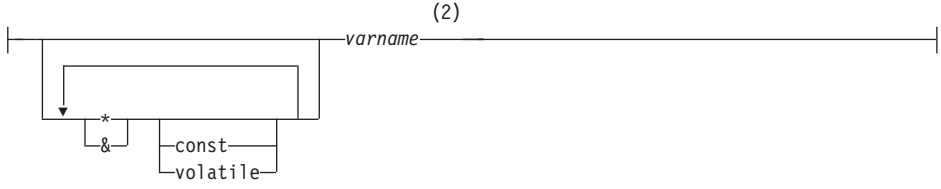
Syntax for Graphic Declaration of Single-Graphic and Null-Terminated Graphic Forms in C and C++

Following is the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.

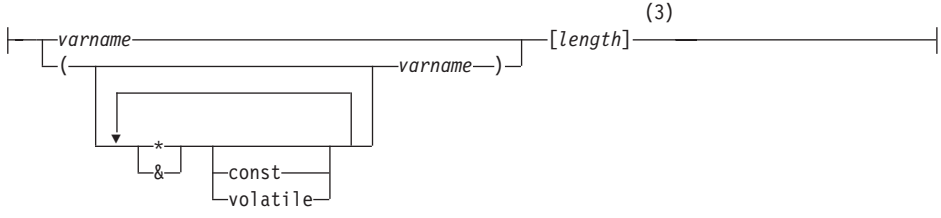
Syntax for Graphic Declaration of Single-Graphic Form and Null-Terminated Graphic Form



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.

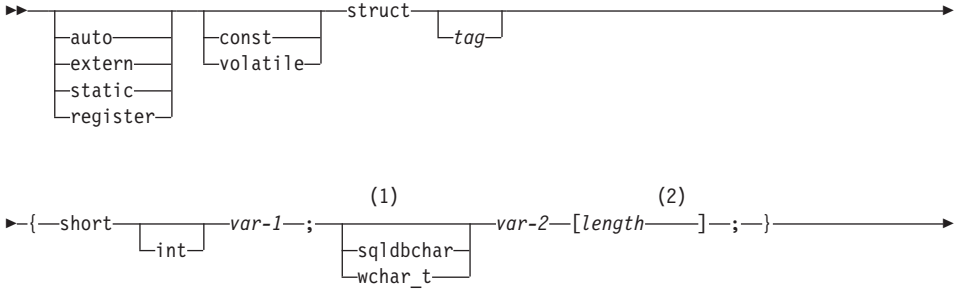
Related concepts:

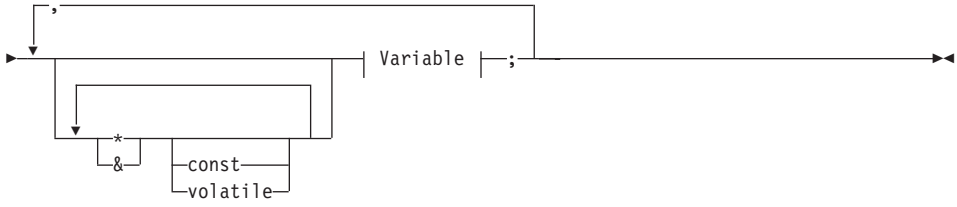
- "Null-Terminated Strings in C and C++" on page 188
- "wchar_t and sqldbchar Data Types in C and C++" on page 193

Syntax for Graphic Declaration of VARGRAPHIC Structured Form in C or C++

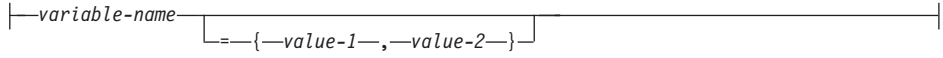
Following is the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

Syntax for Graphic Declaration of VARGRAPHIC Structured Form





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.

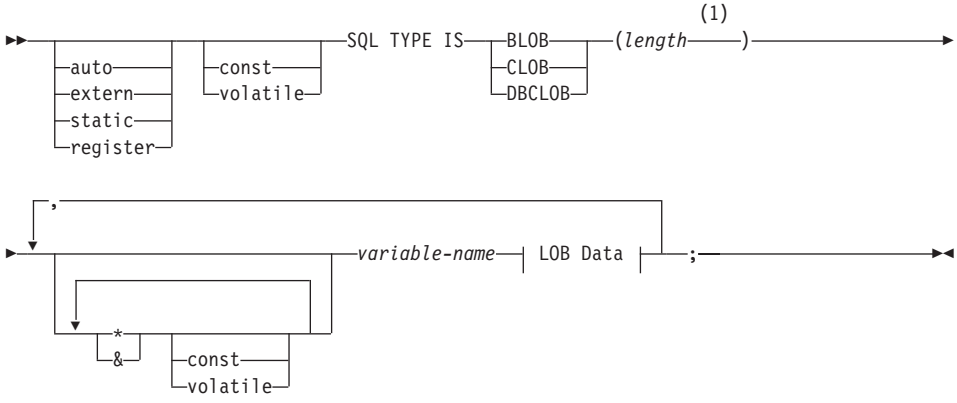
Related concepts:

• "wchar_t and sqldbchar Data Types in C and C++" on page 193

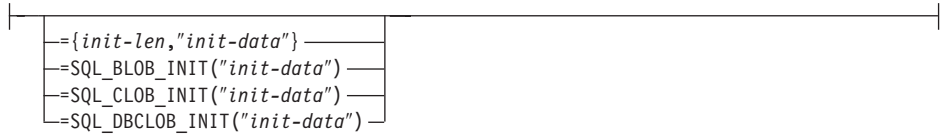
Syntax for Large Object (LOB) Host Variables in C or C++

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");
```

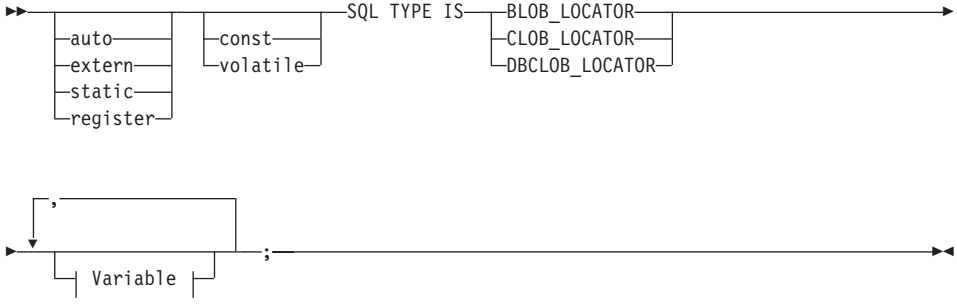
Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

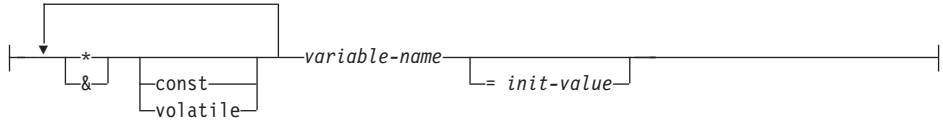
Syntax for Large Object (LOB) Locator Host Variables in C or C++

Following in the syntax for declaring large object (LOB) locator host variables in C or C++.

Syntax for Large Object (LOB) Locator Host Variables in C or C++



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;
```

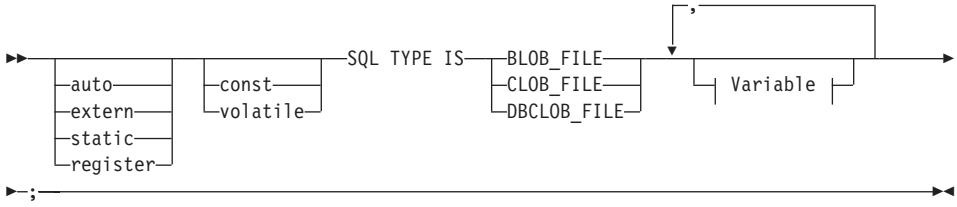
Results in the generation of the following declaration:

```
sqlint32 my_locator;
```

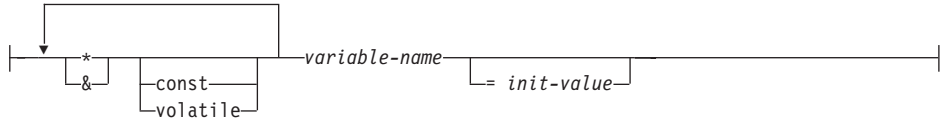
Syntax for File Reference Host Variable Declarations in C or C++

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 187

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 185

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 SQLTYPEs 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 SQLTYPEs 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 SQLTYPEs 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 176
- "wchar_t and sqldbchar Data Types in C and C++" on page 193

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 194
- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 404

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 404

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

Table 13. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description

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] on page 208 (920 or 921)	sql type is clob_file	Descriptor for file containing CLOB data
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

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 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)
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.		

Table 13. SQL Data Types Mapped to C/C++ Declarations (continued)

SQL Column Type[1]	C/C++ Data Type	SQL Column Type Description
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 198

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 198

Related reference:

- "Supported SQL Data Types in C and C++" on page 200

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
DOUBLE (480 or 481)	double	Double-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
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.

Table 14. SQL Data Types Mapped to C/C++ Declarations (continued)

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

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];
```

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.

By default, all applications have a single context that is used for all database access. While this is perfect for a single threaded application, the serialization of SQL statements makes a single context inadequate for a multithreaded application. By using the following DB2 APIs, your application can attach a separate context to each thread and allow contexts to be passed between threads:

- `sqleSetTypeCtx()`
- `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 426

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*
- "sqleSetTypeCtx - Set Application Context Type" 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.

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 210

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 210

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 475

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 220
- “Host Variable Declarations in COBOL” on page 220

Related reference:

- “Syntax for Numeric Host Variables in COBOL” on page 221
- “Syntax for Fixed-Length Character Host Variables in COBOL” on page 222

- "Syntax for Fixed-Length Graphic Host Variables in COBOL" on page 224
- "Syntax for LOB Host Variables in COBOL" on page 225
- "Syntax for LOB Locator Host Variables in COBOL" on page 226
- "Syntax for File Reference Host Variables in COBOL" on page 226

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 220

Related reference:

- "Syntax for Numeric Host Variables in COBOL" on page 221
- "Syntax for Fixed-Length Character Host Variables in COBOL" on page 222
- "Syntax for Fixed-Length Graphic Host Variables in COBOL" on page 224
- "Syntax for LOB Host Variables in COBOL" on page 225
- "Syntax for LOB Locator Host Variables in COBOL" on page 226
- "Syntax for File Reference Host Variables in COBOL" on page 226

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*

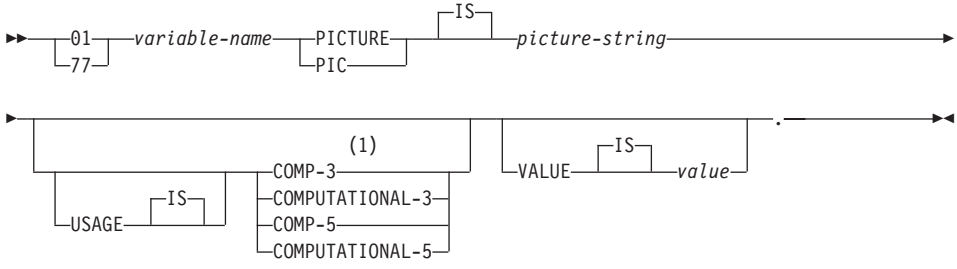
Related reference:

- "Syntax for Numeric Host Variables in COBOL" on page 221
- "Syntax for Fixed-Length Character Host Variables in COBOL" on page 222
- "Syntax for Fixed-Length Graphic Host Variables in COBOL" on page 224
- "Syntax for LOB Host Variables in COBOL" on page 225
- "Syntax for LOB Locator Host Variables in COBOL" on page 226
- "Syntax for File Reference Host Variables in COBOL" on page 226

Syntax for Numeric Host Variables in COBOL

Following is the syntax for numeric host variables.

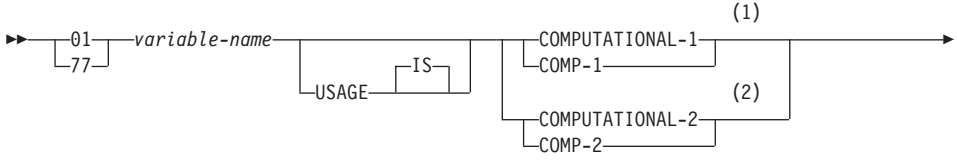
Syntax for Numeric Host Variables in COBOL



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

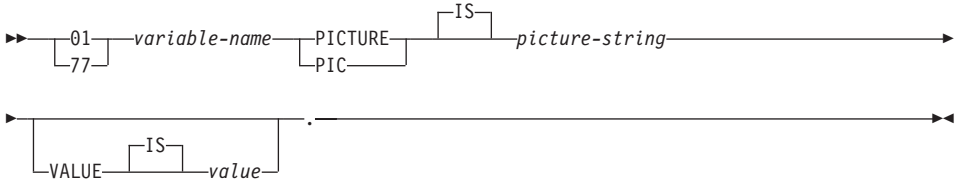
Numeric Host Variable Considerations:

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.

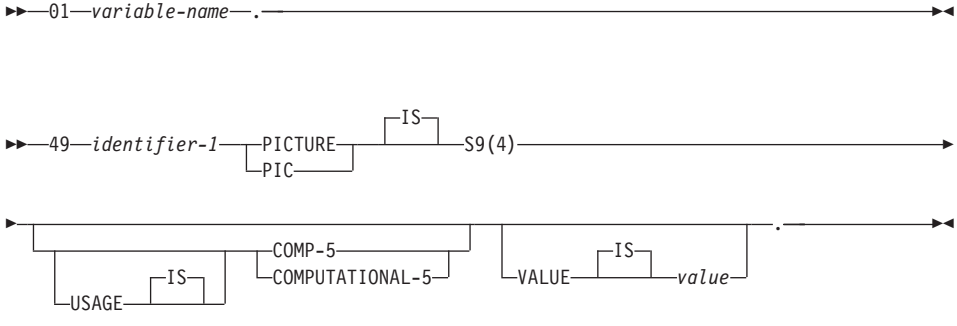
Syntax for Fixed-Length Character Host Variables in COBOL

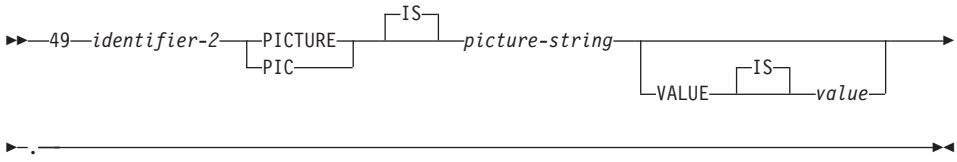
Following is the syntax for character host variables.

Syntax for Character Host Variables in COBOL: Fixed Length



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.
   ```

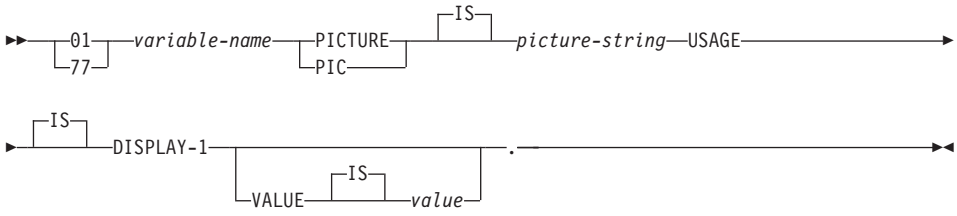
   However, because blanks can be significant in passwords, the `p-word` host variable should be declared as a VARCHAR data item, so that your application can explicitly indicate the significant password length for the CONNECT statement as follows:

   ```
   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.
   ```

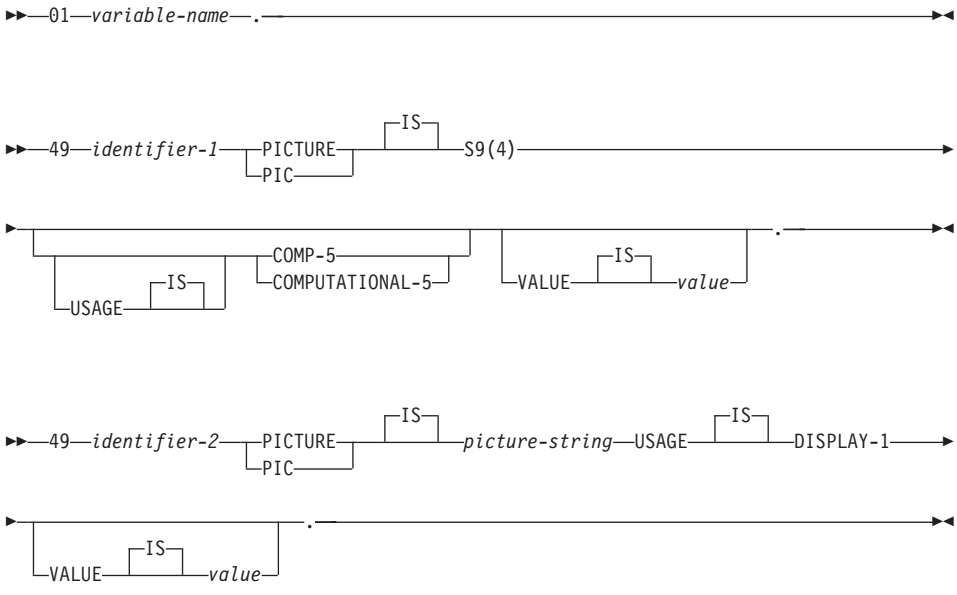
Syntax for Fixed-Length Graphic Host Variables in COBOL

Following is the syntax for graphic host variables.

Syntax for Graphic Host Variables in COBOL: Fixed Length



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.

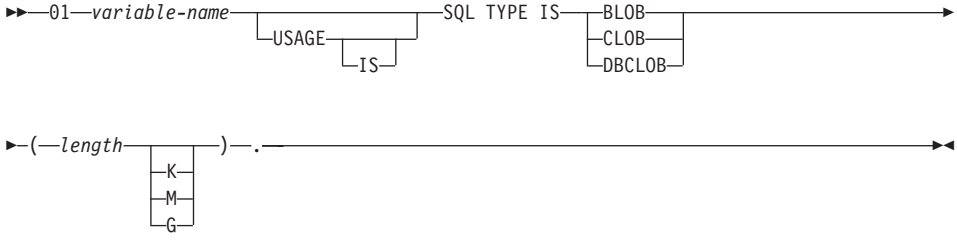
Related concepts:

- "Indicator Tables in COBOL" on page 229

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).
```

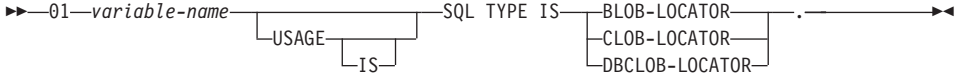
Results in the generation of the following structure:

```
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.
```

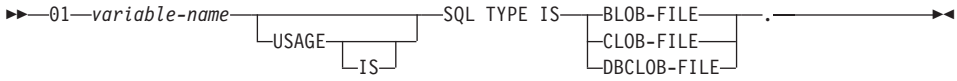
Results in the generation of the following declaration:

```
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 225

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 231

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
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

Table 15. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
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
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

Table 15. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description

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 231

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 231

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 404

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" 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 241

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 239

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 475

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 244
- "Host Variable Declarations in FORTRAN" on page 245
- "Indicator Variables in FORTRAN" on page 247

Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 245
- "Syntax for Character Host Variables in FORTRAN" on page 246
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 248
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 249
- "Syntax for File Reference Host Variables in FORTRAN" on page 249

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 245

Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 245
- "Syntax for Character Host Variables in FORTRAN" on page 246
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 248

- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 249
- "Syntax for File Reference Host Variables in FORTRAN" on page 249

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*

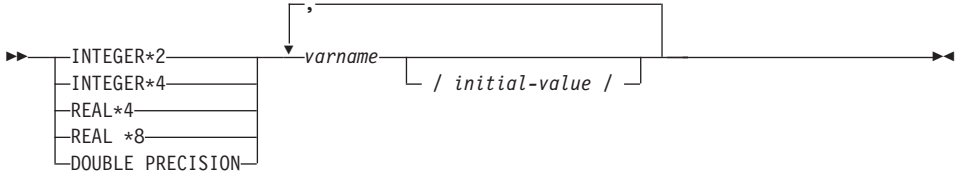
Related reference:

- "Syntax for Numeric Host Variables in FORTRAN" on page 245
- "Syntax for Character Host Variables in FORTRAN" on page 246
- "Syntax for Large Object (LOB) Host Variables in FORTRAN" on page 248
- "Syntax for Large Object (LOB) Locator Host Variables in FORTRAN" on page 249
- "Syntax for File Reference Host Variables in FORTRAN" on page 249

Syntax for Numeric Host Variables in FORTRAN

Following is the syntax for numeric host variables in FORTRAN.

Syntax for Numeric Host Variables in FORTRAN



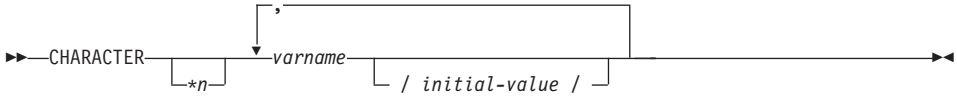
Numeric Host Variable Considerations:

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

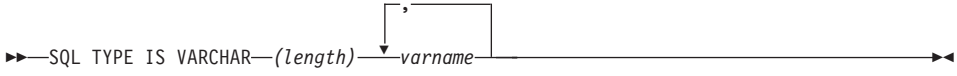
Following is the syntax for fixed-length character host variables.

Syntax for Character Host Variables in FORTRAN: Fixed Length



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
```

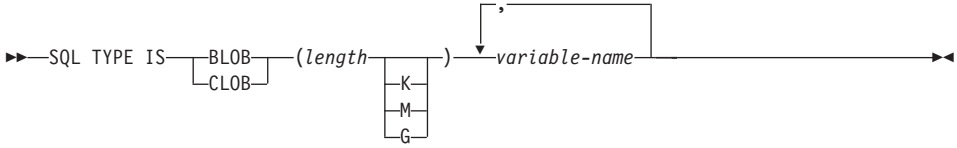
Indicator Variables in FORTRAN

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:

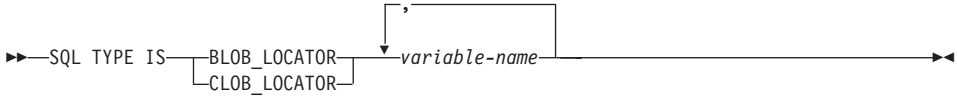
```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
```

```
 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 in FORTRAN



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
```

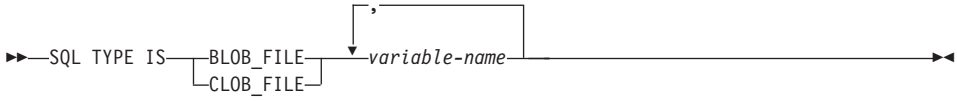
Results in the generation of the following declaration:

```
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 251

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
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

Table 16. SQL Data Types Mapped to FORTRAN Declarations (continued)

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
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 250

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 404

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" in the *Command Reference*

Part 3. Java

Chapter 10. Programming in Java

Programming Considerations for Java

DB2 Universal Database implements two standards-based Java™ programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLj). This chapter provides an overview of JDBC and SQLj programming, but focuses on the aspects specific to DB2. See the DB2 Universal Database Java Web site for links to the JDBC and SQLj specifications.

Related reference:

- "JDBC Samples" in the *Application Development Guide: Building and Running Applications*

- "SQLJ Samples" in the *Application Development Guide: Building and Running Applications*

JDBC and SQLj

The following sections compare JDBC and SQLj, and describe interoperability and session sharing between JDBC and SQLj.

Comparison of SQLj to JDBC

The JDBC API allows you to write Java™ programs that make dynamic SQL calls to databases. SQLj applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLj source files. You must translate an SQLj source file with the SQLj translator before you can compile the resulting Java source code.

JDBC and SQLj Interoperability

The SQLj language provides direct support for static SQL operations that are known at the time the program is written. If some or all of a particular SQL statement cannot be determined until run time, it is a dynamic operation. To perform dynamic SQL operations from an SQLj program, use JDBC. A ConnectionContext object contains a JDBC Connection object, which can be used to create JDBC Statement objects needed for dynamic SQL operations.

Every SQLj ConnectionContext class includes a constructor that takes as an argument a JDBC Connection. This constructor is used to create an SQLj connection context instance that shares its underlying database connection with that of the JDBC connection.

Every SQLj ConnectionContext instance has a `getConnection()` method that returns a JDBC Connection instance. The JDBC Connection returned shares the underlying database connection with the SQLj connection context. It may be used to perform dynamic SQL operations as described in the JDBC API.

Related concepts:

- "Session Sharing Between JDBC and SQLj" on page 258
- "Connection Resource Management in Java" on page 260

Session Sharing Between JDBC and SQLj

The interoperability methods between JDBC and SQLj provide a conversion between the connection abstractions used in SQLj and those used in JDBC. Both abstractions share the same database session; that is, the underlying database connection. Accordingly, calls to methods that affect the session state

on one object will also be reflected in the other object, as it is actually the underlying shared session that is being affected.

JDBC defines the default values for the session state of newly created connections. In most cases, SQLj adopts these default values. However, whereas a newly created JDBC connection has auto commit mode on by default, an SQLj connection context requires the auto commit mode to be specified explicitly upon construction.

Related concepts:

- "JDBC and SQLj Interoperability" on page 258
- "Connection Resource Management in Java" on page 260

Advantages of Java over Other Languages

Programming languages containing embedded SQL are called host languages. Java™ differs from the traditional host languages C, COBOL, and FORTRAN, in ways that significantly affect how it embeds SQL:

- SQLj and JDBC are open standards, enabling you to easily port SQLj or JDBC applications from other standards-compliant database systems to DB2 Universal Database.
- All Java types representing composite data, and data of varying sizes, have a distinguishing value, `null`, which can be used to represent the SQL NULL state, giving Java programs an alternative to NULL indicators that are a fixture of other host languages.
- Java is designed to support programs that are automatically heterogeneously portable (also called "super portable" or simply "downloadable"). Along with Java's type system of classes and interfaces, this feature enables component software. In particular, an SQLj translator written in Java can call components that are specialized by database vendors in order to leverage existing database functions such as authorization, schema checking, type checking, transactional, and recovery capabilities, and to generate code optimized for specific databases.
- Java is designed for binary portability in heterogeneous networks, which promises to enable binary portability for database applications that use static SQL.

SQL Security in Java

By default, a JDBC program executes SQL statements with the privileges assigned to the person who runs the program. In contrast, an SQLj program executes SQL statements with the privileges assigned to the person who created the database package.

Calling the `close()` method of a connection context instance causes the associated JDBC connection instance and the underlying database connection to be closed. Because connection contexts may share the underlying database connection with other connection contexts and/or JDBC connections, it may not be desirable to close the underlying database connection when a connection context is closed. A programmer may want to release the resources maintained by the connection context (for example, statement handles), without actually closing the underlying database connection. To this end, connection context classes also support a `close()` method, which takes a Boolean argument indicating whether or not to close the underlying database connection: the constant `CLOSE_CONNECTION` if the database connection should be closed, and `KEEP_CONNECTION` if it should be retained. The variant of `close()` that takes no arguments is a shorthand for calling `close(CLOSE_CONNECTION)`.

If a connection context instance is not explicitly closed before it is garbage collected, `close(KEEP_CONNECTION)` is called by the finalize method of the connection context. This allows connection-related resources to be reclaimed by the normal garbage collection process while maintaining the underlying database connection for other JDBC and SQLj objects that may be using it. Note that if no other JDBC or SQLj objects are using the connection, the database connection is closed and reclaimed by the garbage collection process.

Both SQLj connection context objects and JDBC connection objects respond to the `close()` method. When writing an SQLj program, it is sufficient to call the `close()` method on only the connection context object: closing the connection context also closes the JDBC connection associated with it. However, it is not sufficient to close only the JDBC connection returned by the `getConnection()` method of a connection context: the `close()` method of a JDBC connection does not cause the containing connection context to be closed, and therefore resources maintained by the connection context are not released until it is garbage collected.

The `isClosed()` method of a connection context returns `true` if any variant of the `close()` method has been called on the connection context instance. If `isClosed()` returns `true`, calling `close()` has no effect, and calling any other method is undefined.

Related concepts:

- "JDBC and SQLj Interoperability" on page 258
- "Session Sharing Between JDBC and SQLj" on page 258

Source and Output Files for Java

Source files have the following extensions:

.java Java™ source files, which require no precompiling. You can compile these files with the `javac` Java compiler included with your Java development environment.

.sqlj SQLj source files, which require translation with the `sqlj` translator. The translator creates:

- One or more `.class` bytecode files
- One `.ser` profile file per connection context

The corresponding output files have the following extensions:

.class JDBC and SQLj bytecode compiled files.

.ser SQLj serialized profile files. You create packages in the database for each profile file with the `db2profc` utility.

Java Class Libraries

DB2 Universal Database provides class libraries for JDBC and SQLj support, which you must provide in your CLASSPATH or include with your applets as follows:

db2jcc.jar
Provides the JDBC Type 4 driver.

db2java.zip
Provides the JDBC driver and JDBC and SQLj support classes, including stored procedure and UDF support.

sqlj.zip
Provides the SQLj translator class files.

runtime.zip
Provides Java™ run-time support for SQLj applications and applets.

Where to Put Java Classes

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

Note: If you update or replace Java routine class files, you must issue a CALL SQLJ.REFRESH_CLASSES() statement to enable DB2® to load the

updated classes. For more information on the CALL SQLJ.REFRESH_CLASSES() statement, see the description of how to update Java classes for routines.

To enable DB2 to find and use your stored procedures and UDFs, you must store the corresponding class files in the *function directory*, which is a directory defined for your operating system as follows:

Unix operating systems
`sqllib/function`

Windows® operating systems
instance_name`\function`, where *instance_name* represents the value of the *DB2INSTPROF* instance-specific registry setting.

For example, the function directory for a Windows NT® server with DB2 installed in the `C:\sqllib` directory, and with no specified *DB2INSTPROF* registry setting, is:

```
C:\sqllib\function
```

If you decide to use individual class files, you must store the class files in the appropriate directory for your operating system. If you declare a class to be part of a Java package, create the corresponding subdirectories in the function directory and place the files in the corresponding subdirectory. For example, if you create a class `ibm.tests.test1` for a Linux system, store the corresponding Java bytecode file (named test1.class) in sqllib/function/ibm/tests.

The JVM that DB2 invokes uses the CLASSPATH environment variable to locate Java files. DB2 adds the function directory and sqllib/java/db2java.zip to the front of your CLASSPATH setting.

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

Related tasks:

- "Updating Java Classes for Runtime" on page 263

Related reference:

- "Maximum Java Interpreter Heap Size configuration parameter - java_heap_sz" in the *Administration Guide: Performance*
- "Java Development Kit Installation Path configuration parameter - jdk_path" in the *Administration Guide: Performance*

Updating Java Classes for Runtime

Procedure:

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

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

Java Packages

To use the class libraries included with DB2 in your own applications, you must include the appropriate `import` *`package`* statements at the top of your source files. You can use the following packages in your Java™ applications:

java.sql.*
The JDBC API included in your JDK. You must import this package in every JDBC and SQLj program.

sqlj.runtime.*
SQLj support included with every DB2® client. You must import this package in every SQLj program.

sqlj.runtime.ref.*
SQLj support included with every DB2 client. You must import this package in every SQLj program.

Host Variables in Java

Arguments to embedded SQL statements are passed through *host variables*, which are variables of the host language that appear in the SQL statement. Host variables have up to three parts:

- A colon prefix, **:**
- An optional parameter mode identifier: IN, OUT, or INOUT
- A Java™ host variable that is a Java identifier for a parameter, variable, or field

The evaluation of a Java identifier does not have side effects in a Java program, so it may appear multiple times in the Java code generated to replace an SQLj clause.

The following query contains the host variable, *:x*, which is the Java variable, field, or parameter *x* visible in the scope containing the query:

```
SELECT  COL1,  COL2  FROM  TABLE1  WHERE  :x  > COL3
```

All host variables specified in compound SQL are input host variables by default. You have to specify the parameter mode identifier OUT or INOUT before the host variable to mark it as an output host variable. For example:

```
#sql {begin compound atomic static
     select count(*) into :OUT count1 from employee;
     end compound}
```

Supported SQL Data Types in Java

The following table shows the Java equivalent of each SQL data type, based on the JDBC specification for data type mappings. The JDBC driver converts the data exchanged between the application and the database using the following mapping schema. Use these mappings in your Java applications and your PARAMETER STYLE JAVA procedures and UDFs.

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

Table 17. SQL Data Types Mapped to Java Declarations

SQL Column Type	Java Data Type	SQL Column Type Description
SMALLINT (500 or 501)	short	16-bit, signed integer
INTEGER (496 or 497)	int	32-bit, signed integer
BIGINT (492 or 493)	long	64-bit, signed integer
REAL (480 or 481)	float	Single precision floating point
DOUBLE (480 or 481)	double	Double precision floating point
DECIMAL(*p*,*s*) (484 or 485)	java.math.BigDecimal	Packed decimal
CHAR(*n*) (452 or 453)	java.lang.String	Fixed-length character string of length *n* where *n* is from 1 to 254

Table 17. SQL Data Types Mapped to Java Declarations (continued)

SQL Column Type	Java Data Type	SQL Column Type Description
CHAR(*n*) FOR BIT DATA	byte[]	Fixed-length character string of length *n* where *n* is from 1 to 254
VARCHAR(*n*) (448 or 449)	java.lang.String	Variable-length character string
VARCHAR(*n*) FOR BIT DATA	byte[]	Variable-length character string
LONG VARCHAR (456 or 457)	java.lang.String	Long variable-length character string
LONG VARCHAR FOR BIT DATA	byte[]	Long variable-length character string
BLOB(*n*) (404 or 405)	java.sql.Blob	Large object variable-length binary string
CLOB(*n*) (408 or 409)	java.sql.Clob	Large object variable-length character string
DBCLOB(*n*) (412 or 413)	java.sql.Clob	Large object variable-length double-byte character string
DATE (384 or 385)	java.sql.Date	10-byte character string
TIME (388 or 389)	java.sql.Time	8-byte character string
TIMESTAMP (392 or 393)	java.sql.Timestamp	26-byte character string

Java Enablement Components

DB2's Java™ enablement has three independent components:

- Support for client applications and applets written in Java using JDBC to access DB2
- Precompile and binding support for client applications and applets written in Java using SQLj to access DB2
- Support for Java UDFs and stored procedures on the server

Related concepts:

- "SQLj Programming" on page 275

Related tasks:

- "Coding JDBC Applications and Applets" on page 268

Application and Applet Support

The sections that follow describe the application and applet support that is provided by the different JDBC drivers.

Application Support in Java with the Type 2 Driver

The following figure shows how a type 2 JDBC application works with DB2. Calls to JDBC are translated to calls to DB2® through Java™ native methods. JDBC requests flow from the DB2 client to the DB2 server.

SQLj applications use this JDBC support, and in addition require the SQLj run-time classes to authenticate and execute any SQL packages that were bound to the database at the precompiling and binding stage.

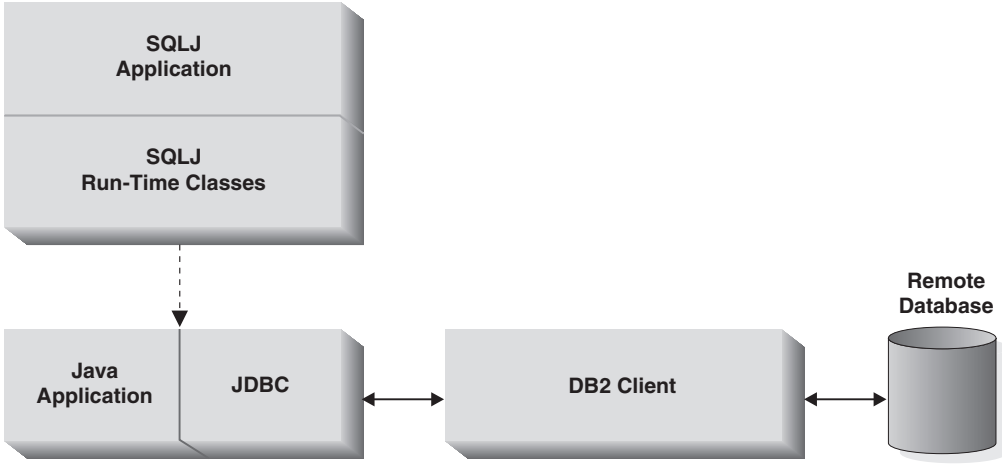


Figure 4. Application Support with the Type 2 Driver

Application and Applet Support in Java with the Type 4 Driver

The following figure shows how a type 4 DB2® JDBC application or applet works. The type 4 application or applet communicates directly with the database.

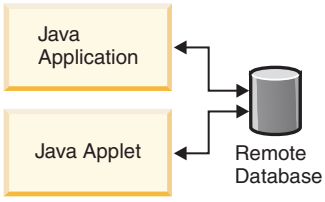


Figure 5. Application and Applet Support with the Type 4 Driver

Applet Support in Java Using the Type 3 Driver

The following figure shows how the JDBC *applet driver*, also known as the *net driver*, works with the JDBC type 3 driver. The type 3 driver consists of a JDBC client and a JDBC server, `db2jd`. The JDBC client driver is loaded on the Web browser along with the applet. When the applet requests a connection to a DB2 database, the client opens a TCP/IP socket to the JDBC server on the machine where the Web server is running. After a connection is set up, the client sends each of the subsequent database access requests from the applet to the JDBC server though the TCP/IP connection. The JDBC server then makes corresponding CLI (ODBC) calls to perform the task. Upon completion, the JDBC server sends the results back to the client through the connection.

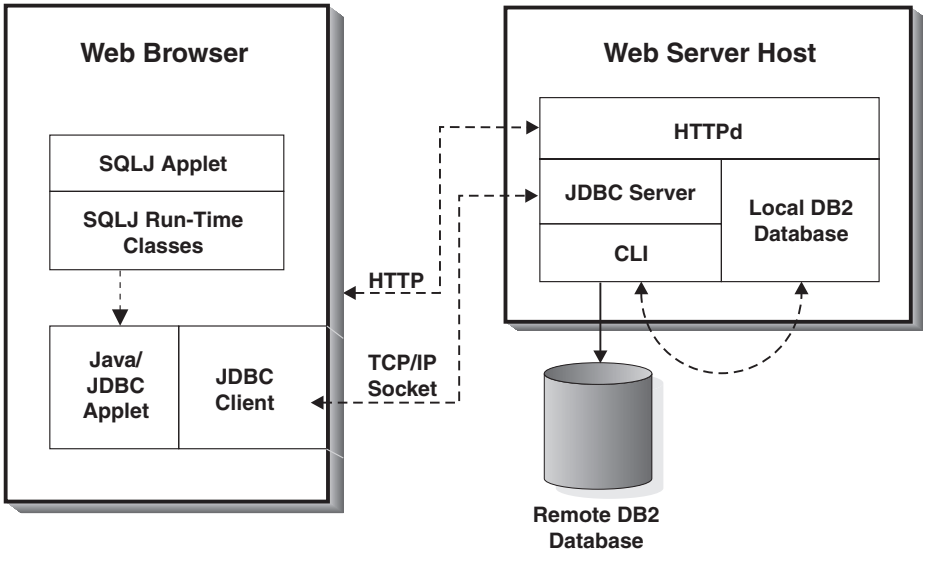


Figure 6. DB2® Java™ Applet Implementation with the JDBC Type 3 Driver

SQLj applets add the SQLj client driver on top of the JDBC client driver, but otherwise work the same as JDBC applets.

Use the db2jstrt command to start the DB2 JDBC server.

JDBC Programming

The sections that follow describe how to create JDBC applications.

Coding JDBC Applications and Applets

JDBC applications and applets typically follow similar program logic.

Procedure:

When you code a JDBC application or applet, you will typically code them to perform the following tasks:

1. Import the appropriate Java packages and classes (java.sql.*).
2. Load the appropriate JDBC driver:
   - For type 2 JDBC, COM.ibm.db2.jdbc.app.DB2Driver for applications
   - For type 3 JDBC, COM.ibm.db2.jdbc.net.DB2Driver for applets.

     Note: The type 3 driver is deprecated in Version 8.
   - For type 4 JDBC, com.ibm.db2.jcc.DB2Driver for both applications and applets.
3. Connect to the database, specifying the location with a URL as defined in the JDBC specification and using the db2 subprotocol.

   The type 3 and 4 drivers require you to provide the user ID, password, host name and a port number. For the type 3 driver, the port number is the applet server port number. For the Type 4 driver the port number is the DB2 server port number. The type 2 driver implicitly uses the default value for user ID and password from the DB2 client catalog, unless you explicitly specify alternative values.
4. Pass SQL statements to the database.
5. Receive the results.
6. Close the connection.

After coding your program, compile it as you would any other Java program. You don't need to perform any special precompile or bind steps.

Related concepts:

• "Example of a JDBC Program" on page 269

JDBC Specification

Whether your application or applet uses JDBC or SQLj, you need to familiarize yourself with the JDBC specification, which is available from Sun

Microsystems. See the DB2 Java™ Web site for links to JDBC and SQLj resources. This specification describes how to call JDBC APIs to access a database and manipulate data in that database.

Related concepts:

- "Java Enablement Components" on page 265
- "JDBC 2.1" on page 272

Example of a JDBC Program

Every JDBC program must perform the following steps:

1. Import the JDBC package.

   Every JDBC and SQLj program must import the JDBC package.
2. Declare a Connection object.

   The Connection object establishes and manages the database connection.
3. Set the database URL variable.

   The DB2 application driver accepts URLs that take the form of:

   `jdbc:db2:`*`database-name`*
4. Connect to the database.

   The `DriverManager.getConnection()` method is most often used with the following parameters:
   - `getConnection(String url)`, which establishes a connection to the database specified by a URL, and uses the default user ID and password.
   - `getConnection(String url, String userid, String password)`, which establishes a connection to the database specified by a URL, and uses the user ID and password that are specified by `userid` and `password` respectively.

   Note: All JDBC sample programs use the `Db` class defined in the **Util.java** program to perform the connection. You can reuse the `Db` class in the **Util.java** program to connect to a database. See the JDBC sample programs for more information.

The JDBC sample **TutMod.java** shows some basic database modifications use include INSERT, UPDATE, and DELETE statements. Another JDBC sample, **TbMod.java**, is a more comprehensive sample that shows how to insert, update, and delete table data. This sample shows many possible ways to modify table data.

Example of an INSERT Statement

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
  "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) ");
stmt.close();
```

Example of an UPDATE Statement

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
  "UPDATE staff " +
  "  SET salary = salary + 1000 " +
  "  WHERE id >= 310 AND dept = 84");
stmt.close();
```

Example of a DELETE Statement

```
Statement stmt = con.createStatement();
stmt.executeUpdate("DELETE FROM staff " +
                   "  WHERE id >= 310 AND salary > 20000");
stmt.close();
```

Related samples:

- "TbMod.java -- How to modify table data (JDBC)"
- "TbMod.out -- HOW TO MODIFY TABLE DATA (JDBC)"
- "TutMod.java -- Modify data in a table (JDBC)"
- "TutMod.out -- HOW TO MODIFY DATA IN A TABLE (JDBC)"

Distribution of JDBC Applications Using the Type 2 Driver

Distribute your JDBC application as you would any other Java™ application. As the application uses the DB2® client to communicate with the DB2 server, you have no special security concerns; authority verification is performed by the DB2 client.

To run your application on a client machine, you must install on that machine:

- A Java Virtual Machine (JVM), which you need to run any Java code
- A DB2 client, which also includes the DB2 JDBC driver

To build your application, you must also install the JDK for your operating system.

Related tasks:

- "Building JDBC Applications" in the *Application Development Guide: Building and Running Applications*

Distribution and Running of Type 4 Driver JDBC Applets

Like other Java™ applets, you distribute your JDBC applet over the network (intranet or Internet). Typically you would embed the applet in a hypertext markup language (HTML) page. For example, to call the sample applet `DB2Applt.java`, (provided in `sqllib/samples/java`), you might use the following `<APPLET>` tag:

```
<applet code="DB2Applt.class" width=325 height=275 archive="db2jcc.jar">
   <param name="server" value="db_server">
   <param name="port" value="50006">
</applet>
```

To run your applet, you need only a Java-enabled Web browser on the client machine (that is, you do not need to install a DB2® client on the client machine). When you load your HTML page, the applet tag instructs your browser to download the Java applet and the db2jcc.jar class library, which includes the DB2 JDBC driver implemented by the `com.ibm.db2.jcc.DB2Driver` class. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes separate communications with the DB2 database through the JDBC applet server running on the Web server.

Note: To ensure that the Web browser downloads db2jcc.jar from the server, ensure that the CLASSPATH environment variable on the client does *not* include db2jcc.jar. Your applet may not function correctly if the client uses a local version of db2jcc.jar.

Related tasks:

- "Building JDBC Applets" in the *Application Development Guide: Building and Running Applications*

Exceptions Caused by Mismatched db2java.zip Files When Using the JDBC Type 3 Driver

If you are using the type 3 JDBC driver, it is essential that the db2java.zip file used by the Java™ applet be at the same FixPak level as the JDBC applet server. Under normal circumstances, db2java.zip is loaded from the Web Server where the JDBC applet server is running. This ensures a match. If, however, your configuration has the Java applet loading db2java.zip from a different location, a mismatch can occur. Prior to DB2® Version 7.1 FixPak 2, this could lead to unexpected failures. As of DB2 Version 7.1 FixPak 2, matching FixPak levels between the two files is strictly enforced at connection time. If a mismatch is detected, the connection is rejected, and the client receives one of the following exceptions:

- If db2java.zip is at DB2 Version 7.1 FixPak 2 or later:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0621E  Unsupported JDBC server configuration.
```

- If db2java.zip is prior to DB2 Version 7.1 FixPak 2:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0601E  Invalid statement handle or statement is closed.
SQLSTATE=S1000
```

If a mismatch occurs, the JDBC applet server logs one of the following messages in the jdbcerr.log file:

- If the JDBC applet server is at DB2 Version 7.1 FixPak 2 or later:

```
jdbcFSQLConnect: JDBC Applet Server and client (db2java.zip)
versions do not match. Unable to proceed with connection., einfo= -111
```

- If the JDBC applet server is prior to DB2 Version 7.1 FixPak 2:

```
jdbcServiceConnection(): Invalid Request Received., einfo= 0
```

Note: Because the type 4 driver has no JDBC server component, the type of mismatch described above cannot occur. It is recommended that you modify your applications to use the type 4 driver.

JDBC 2.1

JDBC Version 2.1 from Sun has two defined parts: the **core API**, and the **Optional Package API**. For information on the JDBC specification, see the DB2 Universal Database Java™ Web site.

Related concepts:

- "JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 2 Driver" on page 272
- "JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 2 Driver" on page 273
- "JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 4 Driver" on page 273
- "JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 4 Driver" on page 275

Related tasks:

- "Setting Up the Java Environment" in the *Application Development Guide: Building and Running Applications*

JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 2 Driver

The DB2 type 2 JDBC driver supports the JDBC 2.1 core API, however, it does not support all of the features defined in the specification. The DB2 JDBC driver does *not* support the following features:

- Updatable ResultSets
- New SQL types (Array, Ref, Java Object, Struct)
- Customized SQL type mapping

- Scrollable sensitive ResultSets (scroll type of ResultSet.TYPE_SCROLL_SENSITIVE)

Related concepts:

- "JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 2 Driver" on page 273

JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 4 Driver

The DB2 type 4 JDBC driver supports the JDBC 2.1 core API, however, it does not support all of the features defined in the specification. The DB2 JDBC driver does *not* support the following features:

- Updatable ResultSets
- New SQL types (Array, Ref, Java Object, Struct)
- Customized SQL type mapping

JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 2 Driver

The DB2 JDBC type 2 driver supports the following features of the JDBC 2.1 Optional Package API:

- Java™ Naming and Directory Interface (JNDI) for Naming Databases

  DB2® provides the following support for the Javing Naming and Directory Interface (JNDI) for naming databases:

  javax.naming.Context

  This interface is implemented by `COM.ibm.db2.jndi.DB2Context`, which handles the storage and retrieval of DataSource objects. To support persistent associations of logical data source names to physical database information, such as database names, these associations are saved in a file named .db2.jndi. For an application, the file resides (or is created if none exists) in the directory specified by the USER.HOME environment variable. For an applet, you must create this file in the root directory of the web server to facilitate the `lookup()` operation. Applets do not support the `bind()`, `rebind()`, `unbind()` and `rename()` methods of this class. Only applications can bind DataSource objects to JNDI.

  javax.sql.DataSource

  This interface is implemented by `COM.ibm.db2.jdbc.DB2DataSource`. You can save an object of this class in any implementation of `javax.naming.Context`. This class also makes use of connection pooling support.

  `DB2DataSource` supports the following methods:

  – public void setDatabaseName(String databaseName)
  – public void setServerName(String serverName)
  – public void setPortNumber(int portNumber)

javax.naming.InitialContextFactory
This interface is implemented by COM.ibm.db2.jndi.DB2InitialContextFactory, which creates an instance of DB2Context. Applications automatically set the value of the JAVA.NAMING.FACTORY.INITIAL environment variable to COM.ibm.db2.jndi.DB2InitialContextFactory To use this class in an applet, call InitialContext() using the following syntax:

```
Hashtable env = new Hashtable( 5 );
env.put( "java.naming.factory.initial",
         "COM.ibm.db2.jndi.DB2InitialContextFactory" );
Context ctx = new InitialContext( env );
```

- Connection Pooling

  DB2ConnectionPoolDataSource and DB2PooledConnection provide the hooks necessary for you to implement your own connection pooling module, as follows:

  javax.sql.ConnectionPoolDataSource
  This interface is implemented by COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource, and is a factory of COM.ibm.db2.jdbc.DB2PooledConnection objects.

  javax.sql.PooledConnection
  This interface is implemented by COM.ibm.db2.jdbc.DB2PooledConnection.

- Java Transaction APIs (JTA)

  DB2 supports the Java Transaction APIs (JTA) through the DB2 JDBC type 2 driver. DB2 does not provide JTA support with the DB2 JDBC type 3 or type 4 drivers.

  javax.sql.XAConnection
  This interface is implemented by COM.ibm.db2.jdbc.DB2XAConnection.

  javax.sql.XADataSource
  This interface is implemented by COM.ibm.db2.jdbc.DB2XADataSource, and is a factory of COM.ibm.db2.jdbc.DB2PooledConnection objects.

  javax.transactions.xa.XAResource
  This interface is implemented by COM.ibm.db2.jdbc.app.DBXAResource.

  javax.transactions.xa.Xid
  This interface is implemented by COM.ibm.db2.jdbc.DB2Xid.

Related concepts:

- "JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 2 Driver" on page 272

JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 4 Driver

The DB2 JDBC type 4 driver supports the following features of the JDBC 2.1 Optional Package API:

- Java™ Naming and Directory Interface (JNDI) for Naming Databases

  DB2® provides the following support for the Javing Naming and Directory Interface (JNDI) for naming databases:

  javax.sql.DataSource

  This interface is implemented by `com.ibm.db2.jcc.DB2SimpleDataSource`. You can save an object of this class in any implementation of `javax.naming.Context`.

  `DB2SimpleDataSource` supports the following methods:
  - public void setDatabaseName(String databaseName)
  - public void setServerName(String serverName)
  - public void setPortNumber(int portNumber)
  - public void setDescription(String description)
  - public void setLoginTimeout(int seconds)
  - public void setLogWriter(java.io.PrintWriter logWriter)
  - public void setPassword(String password)
  - public void setUser(String user)
  - public void setDriverType(int driverType)

  Note: The only type currently supported is 4; this type must be set explicitly by the application following the construction of a new `DB2SimpleDataSource`.

SQLj Programming

The sections that follow describe how to create SQLj applications.

SQLj Programming

DB2® SQLj support is based on the SQLj ANSI standard. Refer to the DB2 Java™ Web site for a pointer to the ANSI Web site and other SQLj resources. The sections that follow provide an overview of SQLj programming and information that is specific to DB2 SQLj support.

The following kinds of SQL constructs may appear in SQLj programs:

- Queries; for example, SELECT statements and expressions
- SQL Data Change Statements (DML); for example, INSERT, UPDATE, DELETE
- Data Statements; for example, FETCH, SELECT..INTO
- Transaction Control; for example, COMMIT, ROLLBACK, and so on

- Data Definition Language (DDL, also known as Schema Manipulation Language); for example, CREATE, DROP, ALTER
- Calls to stored procedures; for example, CALL MYPROC(:x, :y, :z)
- Invocations of functions; for example, VALUES(MYFUN(:x))

Related concepts:

- "DB2 Support for SQLj" on page 276
- "DB2 Restrictions on SQLj" on page 277

DB2 Support for SQLj

DB2® SQLj support is provided by the DB2 Application Development Client. Along with the JDBC support provided by the DB2 client, DB2 SQLj support allows you to create, build, and run embedded SQL for Java™ applications, applets, stored procedures and user-defined functions (UDFs). These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

The SQLj support provided by the DB2 Application Development Client includes:

- The SQLj translator, `sqlj`, which replaces embedded SQL statements in the SQLj program with Java source statements and generates a serialized profile containing information about the SQL operations found in the SQLj program. The SQLj translator uses the `sqllib/java/sqlj.zip` file.
- The SQLj run-time classes, available in `sqllib/java/runtime.zip`.
- The DB2 SQLj profile customizer, `db2profc`, which precompiles the SQL statements stored in the generated profile and generates a package in the DB2 database.
- The DB2 SQLj profile printer, `db2profp`, which prints the contents of a DB2 customized profile in plain text.
- The SQLj profile auditor installer, `profdb`, which installs (or uninstalls) debugging class-auditors into an existing set of binary profiles. Once installed, all `RTStatement` and `RTResultSet` calls made during application run time are logged to a file (or standard output), which can then be inspected to verify expected behavior and trace errors. Note that only those calls made to the underlying `RTStatement` and `RTResultSet` call interface at run time are audited.
- The SQLj profile conversion tool, `profconv`, which converts a serialized profile instance to class bytecode format. Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. As a work-around, you need to run this utility to perform the conversion.

For more information on the SQLj run-time classes, refer to the DB2 Java Web site.

Related concepts:

- "DB2 Restrictions on SQLj" on page 277

Related reference:

- "db2profc - DB2 SQLj Profile Customizer" in the *Command Reference*
- "db2profp - DB2 SQLj Profile Printer" in the *Command Reference*

DB2 Restrictions on SQLj

When you create DB2 applications with SQLj, you should be aware of the following restrictions:

- DB2® SQLj support adheres to standard DB2 Universal Database restrictions on issuing SQL statements.
- A positioned UPDATE and DELETE statement is not a valid sub-statement in a Compound SQL statement.
- The precompile option "DATETIME" is not supported. Only the date and time formats of the International Standards Organization are supported.
- The precompile option "PACKAGE USING package-name" specifies the name of the package that is to be generated by the translator. If a name is not entered, the name of the profile (minus extension and folded to uppercase) is used. Maximum length is 8 characters. Since the SQLj profile name has the suffix _SJProfile*N*, where *N* is the profile key number, the profile name will always be longer than 8 characters. The default package name will be constructed by concatenating the first (8 - *pfKeyNumLen*) characters of the profile number and the profile key number, where *pfKeyNumLen* is the length of the profile key number in the profile name. If the length of the profile key number is longer than 7, the last 7 digits will be used without any warnings. For example:

```
profile name                    default package name
--------------------            --------------------
App_SJProfile1                  App_SJP1
App_SJProfile123                App_S123
App_SJProfile1234567            A1234567
App_SJProfile12345678           A2345678
```

- When a `java.math.BigDecimal` host variable is used, the precision and scale of the host variable is not available during the translation of the application. If the precision and scale of the decimal host variable is not obvious from the context of the statement in which it is used, the precision and scale can be specified using a CAST.
- A Java™ variable with type `java.math.BigInteger` cannot be used as a host variable in an SQL statement.

Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. You will get the following error message when trying to load the applet Applt in those browsers:

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

As a work-around, there is a utility which converts a serialized profile into a profile stored in Java class format. The utility is a Java class called `sqlj.runtime.profile.util.SerProfileToClass`. It takes a serialized profile resource file as input and produces a Java class containing the profile as output. Your profile can be converted using the following command:

```
profconv Applt_SJProfile0.ser
```

or

```
java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

The class `Applt_SJProfile0.class` is created as a result. Replace all profiles in `.ser` format used by the applet with profiles in `.class` format.

For an SQLj applet, you need both `db2java.zip` and `runtime.zip` files. If you choose not to package all your applet classes, classes in `db2java.zip` and `runtime.zip` into a single Jar file, put both `db2java.zip` and `runtime.zip` (separated by a comma) into the archive parameter in the ″applet″ tag. For those browsers that do not support multiple zip files in the archive tag, specify `db2java.zip` in the archive tag, and unzip `runtime.zip` with your applet classes in a working directory that is accessible to your web browser.

Embedded SQL Statements in Java

Static SQL statements in SQLj appear in *SQLj clauses*. SQLj clauses are the mechanism by which SQL statements in Java™ programs are communicated to the database.

The SQLj translator recognizes SQLj clauses and SQL statements because of their structure, as follows:

- SQLj clauses begin with the token `#sql`
- SQLj clauses end with a semicolon

The simplest SQLj clauses are *executable clauses* and consist of the token `#sql` followed by an SQL statement enclosed in braces. For example, the following SQLj clause may appear wherever a Java statement may legally appear. Its purpose is to delete all rows in the table named `TAB`:

```
#sql { DELETE FROM TAB };
```

In an SQLj executable clause, the tokens that appear inside the braces are SQL tokens, except for the host variables. All host variables are distinguished by the colon character so the translator can identify them. SQL tokens never occur outside the braces of an SQLj executable clause. For example, the

following Java method inserts its arguments into an SQL table. The method body consists of an SQLj executable clause containing the host variables *x*, *y*, and *z*:

```
void m (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

Do not intitialize static SQL statements in a loop. One physical statement must exist for each static SQL statement. For example, replace:

```
for( int i=0; i<2; i++){
   #sql [ctx] itr[i] = { SELECT id, name FROM staff };
}
```

with the following:

```
int i=0;
#sql [ctx] itr[i] = { SELECT id, name FROM staff };
i=1;
#sql [ctx] itr[i] = { SELECT id, name FROM staff };
```

In general, SQL tokens are case insensitive (except for identifiers delimited by double quotation marks), and can be written in upper, lower, or mixed case. Java tokens, however, are case sensitive. For clarity in examples, case insensitive SQL tokens are uppercase, and Java tokens are lowercase or mixed case. The lowercase `null` is used to represent the Java null value, and the uppercase `NULL` to represent the SQL null value.

Related concepts:

• "Iterator Declarations and Behavior in SQLj" on page 279

Iterator Declarations and Behavior in SQLj

Unlike SQL statements that retrieve data from a table, applications that perform positioned UPDATE and DELETE operations, or that use iterators with `holdability` or `returnability` attributes, require two Java™ source files. Declare the iterator as `public` in one source file, appending the `with` and `implements` clause as appropriate.

To set the value of the `holdability` or `returnability` attribute, you must declare the iterator using the `with` clause for the corresponding attribute. The following example sets the `holdability` attribute to `true` for the iterator `WithHoldCurs`:

```
#sql public iterator WithHoldCurs with (holdability=true) (String EmpNo);
```

Iterators that perform positioned updates require an `implements` clause that implements the `sqlj.runtime.ForUpdate` interface. For example, suppose that you declare iterator DelByName like this in `file1.sqlj`:

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

You can then use the translated and compiled iterator in a different source file. To use the iterator:

1. Declare an instance of the generated iterator class
2. Assign the SELECT statement for the positioned UPDATE or DELETE to the iterator instance
3. Execute positioned UPDATE or DELETE statements using the iterator

To use DelByName for a positioned DELETE in `file2.sqlj`, execute statements like those in the following example:

```
{
   DelByName deliter; // Declare object of DelByName class
   String enum;
1  #sql deliter = { SELECT EMPNO FROM EMP WHERE WORKDEPT='D11'};
   while (deliter.next())
   {
2     enum = deliter.EmpNo(); // Get value from result table
3     #sql { DELETE WHERE CURRENT OF :deliter };
      // Delete row where cursor is positioned
   }
}
```

Notes:

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 *deliter*.
2. This statement positions the iterator to the next row to be deleted.
3. This SQLj clause performs the positioned DELETE.

Related concepts:

• "Example of Iterators in an SQLj Program" on page 280

Example of Iterators in an SQLj Program

The SQLj sample, **TbRead.sqlj**, uses static SQL to retrieve data from the DEPARTMENT table of the DB2 **sample** database. This sample shows two types of iterators:

- Named binding to columns

  This iterator declares column data types and names, and returns the values of the columns according to column name. The following example is demonstrated by the `selectUsingNamedBindingToColumns()` function of the **TbRead.sqlj** sample:

```
#sql iterator Named_Iterator(String deptnumb, String deptname);
Named_Iterator namedIter = null;

// declare a cursor
```

```
#sql namedIter = {SELECT deptno as deptnumb, deptname
                    FROM department
                    WHERE admrdept = 'A00'};

// retrieve the values of the columns according to column name
while (namedIter.next())
{
  System.out.println( namedIter.deptnumb() + ", "+ namedIter.deptname() );
}

// close the cursor
namedIter.close();
```

- Positional binding to columns

  This iterator declares column data types, and returns the values of the columns by column position. The following example is demonstrated by the `selectUsingPositionalBindingToColumns()` function of the **TbRead.sqlj** sample:

```
#sql iterator Positioned_Iterator(String, String);
Positioned_Iterator posIter;
String deptnumb = "";
String deptname = "";

// delcare cursor
#sql posIter = {SELECT deptno as deptnumb, deptname
                  FROM department
                  WHERE admrdept = 'A00'};

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

while (!posIter.endFetch())
{
  System.out.println( deptnumb + ", " + deptname );
  #sql {FETCH :posIter INTO :deptnumb, :deptname};
}

// close the cursor
posIter.close();
```

Related samples:

- "TbRead.out -- HOW TO READ TABLE DATA (SQLJ)"
- "TbRead.sqlj -- How to read table data (SQLj)"

Calls to Routines in SQLj

Databases may contain *procedures*, *user-defined functions*, and *user-defined methods*. Procedures, user-defined functions, and user-defined methods are named schema objects that execute in the database. An SQLj executable clause appearing as a Java™ statement may call a procedure by means of a CALL statement like the following:

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

Procedures may have IN, OUT, or INOUT parameters. In the above case, the value of host variable *myarg* is changed by the execution of that clause. An SQLj executable clause may call a function by means of the SQL `VALUES` construct. For example, assume a function `F` that returns an integer. The following example illustrates a call to that function that then assigns its result to Java local variable *x*:

```
{
    int x;
    #sql x = { VALUES( F(34) ) };
}
```

Related concepts:

- "References to Functions" in the *Application Development Guide: Programming Server Applications*
- "Routine Names and Paths" in the *Application Development Guide: Programming Server Applications*
- "References to Procedures" in the *Application Development Guide: Programming Server Applications*

Related tasks:

- "Building SQLJ Routines" in the *Application Development Guide: Building and Running Applications*
- "Invoking a Procedure" in the *Application Development Guide: Programming Server Applications*
- "Invoking Routines" in the *Application Development Guide: Programming Server Applications*
- "Invoking a UDF" in the *Application Development Guide: Programming Server Applications*
- "Invoking a Table Function" in the *Application Development Guide: Programming Server Applications*

Related reference:

- "SQLJ Samples" in the *Application Development Guide: Building and Running Applications*

Example of Compiling and Running an SQLj Program

Assume that you have an SQLj program called `MyClass`. To run this program, you would do the following:

1. Translate the SQLj (embedded SQL for Java) source file, `MyClass.sqlj`, with the SQLj translator to generate the Java™ source file, `Myclass.java`.

The translator also creates the profiles `MyClass_SJProfile0.ser`, `MyClass_SJProfile1.ser`, ... (one profile for each connection context):

```
sqlj MyClass.sqlj
```

When you use the `sqlj` translator without specifying an `sqlj.properties` file, the translator uses the following values:

```
sqlj.url=jdbc:db2:sample
sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
sqlj.online=sqlj.semantics.JdbcChecker
sqlj.offline=sqlj.semantics.OfflineChecker
```

If you do specify an `sqlj.properties` file, make sure the following options are set:

```
sqlj.url=jdbc:db2:dbname
sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
sqlj.online=sqlj.semantics.JdbcChecker
sqlj.offline=sqlj.semantics.OfflineChecker
```

where *dbname* is the name of the database. You can also specify these options on the command line. For example, to specify the database `mydata` when translating `MyClass`, you can issue the following command:

```
sqlj -url=jdbc:db2:mydata MyClass.sqlj
```

Note that the SQLj translator automatically compiles the translated source code into class files, unless you explicitly turn off the compile option with the `-compile=false` clause.

2. Install DB2 SQLj Customizers on generated profiles and create the DB2® packages in the DB2 database *dbname*:

```
db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
  -prepoptions="bindfile using MyClass0.bnd package using MyClass0"
  MyClass_SJProfile0.ser
db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
  -prepoptions="bindfile using MyClass1.bnd package using MyClass1"
  MyClass_SJProfile1.ser
...
```

3. Execute the SQLj program:

```
java MyClass
```

The translator generates the SQL syntax for the database for which the SQLj profile is customized. For example:

```
i = { VALUES (  F(:x) ) };
```

is translated by the SQLj translator and stored as:

```
? = VALUES (F (?))
```

in the generated profile. When connecting to a DB2 Universal Database database, DB2 will customize the VALUE statement into:

```
VALUES(F(?)) INTO ?
```

but when connecting to a DB2 Universal Database for OS/390 and z/OS database, DB2 customizes the VALUE statement into:

```
SELECT F(?) INTO ? FROM SYSIBM.SYSDUMMY1
```

If you run the DB2 SQLj profile customizer, `db2profc`, against a DB2 Universal Database database and generate a bind file, you cannot use that bind file to bind up to a DB2 for OS/390® database when there is a `VALUES` clause in the bind file. This also applies to generating a bind file against a DB2 for OS/390 database and trying to bind with it to a DB2 Universal Database database.

Related tasks:

- "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 SQLJ Routines" in the *Application Development Guide: Building and Running Applications*
- "Building SQLJ Programs" in the *Application Development Guide: Building and Running Applications*

SQLj Translator Options

The SQLj translator supports the same precompile options as the DB2® PRECOMPILE command, with the following exceptions:

```
CONNECT
DISCONNECT
DYNAMICRULES
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
SQLFLAG
SQLRULES
SYNCPOINT
TARGET
WCHARTYPE
```

To print the content of the profiles generated by the SQLj translator in plain text, use the `profp` utility as follows:

```
profp MyClass_SJProfile0.ser
profp MyClass_SJProfile1.ser
  ...
```

To print the content of the DB2 customized version of the profile in plain text, use the db2profp utility as follows, where *dbname* is the name of the database:

```
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile0.ser
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile1.ser
    ...
```

Troubleshooting Java Applications

The sections that follow describe the trace facilities available for Java, and SQLSTATE and SQLCODE values in Java.

Trace Facilities in Java

Both the CLI/ODBC/JDBC trace facility and the DB2 trace facility, db2trc, can be used to diagnose problems related to JDBC or SQLj programs.

Note: The type 4 JDBC driver does not use the CLI/ODBC/JDBC trace facility. Tracing for the type 4 JDBC driver is enabled using the setLogWriter() method on the javax.sql.DataSource API.

You can also install run-time call tracing capability into SQLj programs. The utility operates on the profiles associated with a program. Suppose a program uses a profile called App_SJProfile0. To install call tracing into the program, use the command:

```
profdb App_SJProfile0.ser
```

The profdb utility uses the Java™ Virtual Machine to run the main() method of class sqlj.runtime.profile.util.AuditorInstaller. For more details on usage and options for the AuditorInstaller class, visit the DB2 Java Web site.

Related concepts:

- "CLI/ODBC/JDBC Trace Facility" on page 285
- "CLI and JDBC Trace Files" on page 294

Related reference:

- "db2trc - Trace" in the *Command Reference*

CLI/ODBC/JDBC Trace Facility

This topic discusses the following subjects:

- "DB2 CLI and DB2 JDBC Trace Configuration" on page 286
- "DB2 CLI Trace Options and the db2cli.ini File" on page 287
- "DB2 JDBC Trace Options and the db2cli.ini File" on page 291

The DB2 CLI and DB2® JDBC drivers 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 DB2 JDBC). 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
- TRACEFILENAME
- TRACEPATHNAME
- TRACEFLUSH
- TRACEREFRESHINTERVAL
- TRACECOMM
- TRACETIMESTAMP
- TRACEPIDTID

- TRACEPIDLIST
- TRACETIME
- TRACESTMTONLY

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.

TRACE = 0 | 1

The TRACE keyword determines whether or not any of the other DB2 CLI trace keywords have effect. If this keyword is unset or set to the default value of 0, the DB2 CLI trace facility is disabled. If this keyword is set to 1, the DB2 CLI trace facility is enabled and the other trace keywords are considered.

By itself, the TRACE keyword has little effect except to enable the DB2 CLI trace facility processing. No trace output is generated unless one of the TRACEPATHNAME or TRACEFILENAME keywords is also specified.

TRACEFILENAME = <fully_qualified_trace_file_name>

The fully qualified name of the log file to which all DB2 CLI trace information is written.

If the file does not exist, the DB2 CLI trace facility will attempt to create it. If the file already exists, new trace information for the current session, if any, will be appended to the previous contents of that file.

The TRACEFILENAME keyword option should not be used with multi-process or multithreaded applications as the trace output for all threads or processes will be written to the same log file, and the output for each thread or process will be difficult to decipher. Furthermore, semaphores are used to control access to the shared trace file which could change the behavior of multithreaded applications. There is no default DB2 CLI trace output log file name.

TRACEPATHNAME = <fully_qualified_trace_path_name>

The fully qualified path name of the directory to which all DB2 CLI trace information is written. The DB2 CLI trace facility will attempt to generate a new trace log file each time an application accessing the DB2 CLI interface is run. If the application is multithreaded, a separate trace log file will be generated for each thread. A concatenation of the application process ID and the thread sequence number is automatically used to name trace log files. There is no default path to which DB2 CLI trace output log files are written, and

the path specified must exist at application execution time (the DB2 CLI driver will not create the path).

Note: If both TRACEFILENAME and TRACEPATHNAME are specified, the TRACEFILENAME keyword takes precedence and TRACEPATHNAME will be ignored.

TRACEFLUSH = 0 | <any positive integer>
The TRACEFLUSH keyword specifies how often trace information is written to the DB2 CLI trace log file. By default, TRACEFLUSH is set to 0 and each DB2 CLI 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 CLI trace log file, the TRACEFLUSH keyword can be specified. After n trace entries have been written to the trace log file, the DB2 CLI driver closes the file and then reopens it, appending new trace entries to the end of the file. Each file close and reopen operation incurs significant input/output overhead and can reduce performance considerably. *The smaller the value of the TRACEFLUSH keyword, the greater the impact DB2 CLI tracing has on the performance of the application.*

Setting TRACEFLUSH=1 has the most impact on performance, but will ensure that each entry is written to disk before the application continues to the next statement.

TRACEREFRESHINTERVAL = 0 | <any positive integer>
Setting TRACEREFRESHINTERVAL to a positive integer value n other than the default value of 0 causes the DB2 CLI trace facility to reread the TRACE and TRACEPIDLIST keywords from the db2cli.ini file at the specified interval (every n seconds). The DB2 CLI trace facility then applies those keywords, as appropriate, to the trace that is currently executing.

The remaining DB2 CLI trace configuration keywords determine what information is written to the DB2 CLI trace log files.

TRACECOMM = 0 | 1
Setting TRACECOMM to the default value of 0 means no DB2 client-server communication information will be included in the DB2 CLI trace. Setting TRACECOMM to 1 causes the DB2 CLI trace to show:

- which DB2 CLI functions are processed completely on the client and which DB2 CLI functions involve communication with the server

- the number of bytes sent and received in each communication with the server
- the time spent communicating data between the client and server

TRACETIMESTAMP = 0 | 1 | 2 | 3
Setting TRACETIMESTAMP to a value other than the default of 0 means the current timestamp or absolute execution time is added to the beginning of each line of trace information as it is being written to the DB2 CLI trace log file. Setting TRACETIMESTAMP to 1 prepends the absolute execution time in seconds and milliseconds, followed by a timestamp. Setting TRACETIMESTAMP to 2 prepends the absolute execution time in seconds and milliseconds. Setting TRACETIMESTAMP to 3 prepends the timestamp.

TRACEPIDTID = 0 | 1
Setting TRACEPIDTID to the default value of 0 means process and thread ID information will not be added to each line in the DB2 CLI trace. Setting TRACEPIDTID to 1 means process and thread ID information will be included in the trace.

TRACEPIDLIST = <no value> | <pid1,pid2, pid3,...>
Setting TRACEPIDLIST to its default of no value, or leaving it unset, means all processes accessing the DB2 CLI driver interface will be traced by the DB2 CLI trace facility. Setting TRACEPIDLIST to a list of one or more comma-delimited process ID values will restrict the CLI traces generated to the processes appearing in that list.

TRACETIME = 0 | 1
Setting TRACETIME to its default value of 1, or leaving it unset, means the elapsed time between CLI function calls and returns will be calculated and included in the DB2 CLI trace. Setting TRACETIME to 0 means the elapsed time between CLI function calls and returns will not be calculated and included in the DB2 CLI trace.

TRACESTMTONLY = 0 | 1
Setting TRACESTMTONLY to its default value of 0 means trace information for all DB2 CLI function calls will be written to the DB2 CLI trace log file. Setting TRACESTMTONLY to 1 means only information related to the `SQLExecute()` and `SQLExecDirect()` function calls will be written to the log file. This trace option can be useful in determining the number of times a statement is executed in an 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 DB2 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 DB2 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 DB2 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. Some versions of the DB2 JDBC driver depend 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, DB2 JDBC driver, and DB2 traces:

Internally, some versions of the DB2 JDBC driver make use of the DB2 CLI driver for database access. For example, the Java getConnection() method may be internally mapped by the DB2 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 294

Related reference:

- "SQLSetEnvAttr Function (CLI) - Set Environment Attribute" in the *CLI Guide and Reference, Volume 2*
- "db2trc - Trace" in the *Command Reference*
- "GET CLI CONFIGURATION" in the *Command Reference*
- "UPDATE CLI CONFIGURATION" 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" on page 295

- "CLI Trace File Interpretation" on page 296
- "JDBC Trace File Interpretation" on page 301

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 285

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*

SQLSTATE and SQLCODE Values in Java

If an SQL error occurs, JDBC and SQLj programs throw an `SQLException`. To retrieve the SQLSTATE, SQLCODE, or SQLMSG values for an instance of an `SQLException`, invoke the corresponding instance method as follows:

SQL return code	SQLException method
SQLCODE	`SQLException.getErrorCode()`
SQLMSG	`SQLException.getMessage()`
SQLSTATE	`SQLException.getSQLState()`

For example:

```
int sqlCode=0;      // Variable to hold SQLCODE
String sqlState="00000";  // Variable to hold SQLSTATE

try
{
  // JDBC statements may throw SQLExceptions
  stmt.executeQuery("Your JDBC statement here");

  // SQLj statements may also throw SQLExeptions
  #sql {..... your SQLj statement here ......};
}

/* Here's how you can check for SQLCODEs and SQLSTATE */

catch (SQLException e)
{
  sqlCode = e.getErrorCode() // Get SQLCODE
```

```
  sqlState = e.getSQLState() // Get SQLSTATE

  if (sqlCode == -190 || sqlState.equals("42837"))
  {
    // Your code here to handle SQLCODE -190 or SQLSTATE 42837
  }
  else
  {
    // Your code here to handle other errors
  }
  System.err.println(e.getMessage()) ; // Print the exception
  System.exit(1);         // Exit
}
```

Chapter 11. Java Applications Using WebSphere Application Servers

Web Services

The Internet infrastructure is ready to support the next generation of e-business applications, called Web services. Web services represent the next level of function and efficiency in e-business. Specifically, Web services are enhanced e-business applications that are easier to advertise and easier to discover — by other businesses — because they are described in a more uniform way on the Internet. These new enhancements allow e-business applications to be connected more easily both inside and outside the enterprise.

A Web service is a set of application functions that perform a service for a requester, such as informational or transactional functions. A Web service can be described and published to the network for use by other programs across the network. Examples of publicly available Web services today include a stock quote service, a service to retrieve news from Web news sources, a service to return maps of historic weather events by zip code, currency conversion services, and a service to return highway conditions in California. Because Web services are modular, related Web services can be aggregated into a larger Web service. For example, it is possible to imagine a wireless application composed of separate Web services that obtain stock quotes, subscribe to news services, convert currency, and manage calendars.

Web services expands the Web's audience to include programs as well as humans. Specifically, Web services creates an architecture that makes it possible for software to do what humans do with the Web; namely, access documents and run applications in a general way, without requiring application specific knowledge and client software. An architecture that supports Web services provides the groundwork to realize that goal.

The Web services infrastructure is based on the eXtensible Markup Language (XML). Messages and data flow between a service requester and a service provider using XML. The sections that follow briefly describe Web services, how DB2® data can be dynamically transformed to XML, and the important role that DB2 plays in a Web services world.

Web services provide a level of abstraction that makes it relatively simple to wrap an existing enterprise application and turn it into a Web service. Web services are based on the XML standard data format and data exchange mechanisms, which provide both flexibility and platform independence. With Web services, requesters typically neither know nor care about the underlying implementation of Web services, which can simplify the integration of heterogeneous business processes.

Web services also provide you with a way to make your key business processes accessible to your customers, partners, and suppliers. For example, an airline can provide its airline reservation systems as a Web service to make it easier for its large corporate customers to integrate the service into their travel planning applications. A supplier can make its inventory levels and pricing accessible to its key buyers.

In one possible example, a buyer matches up incoming purchase orders with transportation services:

1. The buyer accesses the local database to select a list of purchase orders.
2. While viewing the detail for a particular purchase order, the buyer selects a transportation service provider from an approved list, a list that is kept in a private registry.
3. For each provider, the buyer receives quotes dynamically using Web services capabilities. Each request is bound and sent to the location specified in the registry and processed by the supplier's Web service. The supplier's Web service takes input about the request, accesses its database and returns the quote to the requester.
4. The buyer then chooses one service provider based on the prices quoted and the selection is added back to the purchase order page to reflect the selection of a shipping service provider.

Web services are likely to become widespread where existing technologies have not. Web services leverage XML for data representation and exchange,

and do not require complex language-dependent mappings and compile time bindings. Web services offer both ease of development and ease of modification. Further, Web services do not mandate tight synchronous relationships between requesters and service providers. This characteristic further simplifies the implementation of Web services in an Internet environment in which it is impossible to tightly control network behavior. The reliance on XML for data exchange and the abundance of existing and emerging tools for Web service technology make it relatively easy to implement your first Web service.

Web Services Architecture

The nature of Web services make them natural components in a service-oriented architecture. In a typical service-oriented architecture, the service providers host a network-accessible software module, or Web service. A service provider defines a service description for a Web service, and publishes it to a service registry. A service requester uses a find operation to retrieve the service description from the registry, and uses the service description to bind with the service provider and invoke or interact with the Web service implementation.

In simple terms, a Web service is created by wrapping an application in such a way that it can be accessed using standard XML messages, which are themselves wrapped in such a way that masks the underlying transport protocol. The service can be publicized by being registered in a standard-format registry. This registry makes it possible for other people or applications to find and use the service.

The pieces of the Web services architecture include:

- A Web service (a general term used to describe software that can be invoked over the Web)
- Application-specific messages that are sent in standard XML document formats conforming to the corresponding service description.
- The XML messages are contained in *Simple Object Access Protocol* (SOAP) envelopes. SOAP is an application invocation protocol that defines a simple protocol for exchanging information encoded as XML messages.

  Because SOAP makes no assumptions on the implementation of the endpoints, service requester needs only to create an XML request, send it to a service provider, and understand the XML response that comes back. The DB2® implementation is hidden from the requester.

  A SOAP request consists of the envelope itself, which contains the namespaces used by the rest of the SOAP message, an optional header, and the body, which can be a remote procedure call (RPC) or an XML document.

SOAP builds on existing Internet standards such as HTTP and XML, but can be used with any network protocol, programming language, or data encoding model. For example, you can send SOAP messages over IBM® MQSeries, FTP or even as mail messages.

- The logical interface and the service implementation are described by the *Web Services Description Language* (WSDL). WSDL is an XML vocabulary used to automate the details involved in communicating between Web services applications. There are three pieces to WSDL: a data type description (XML Schema), an interface description, and binding information. The interface description is typically used at development time, and the binding information can be used at either development or execution time to invoke a particular service at the specified location. The service description is crucial to making the Web services architecture loosely coupled and reducing the amount of required shared understanding and custom programming between service providers and service requesters.
- To enable service requesters to find your Web service, you can publish descriptive information, such as taxonomy, ownership, business name, business type and so on, via a registry that adheres to the Uniform Description, Discovery and Integration (UDDI) specification or into some other XML registry. The UDDI information can include a pointer to WSDL interfaces, the binding information, as well as the actual business name (the name that makes the purpose of the Web service understandable to users). A UDDI registry is searchable by programs, which enables a service requester to bind to a UDDI provider to find out more information about a service before actually using it.
- The ability to compose Web services together is provided by *Web Services Flow Language* (WSFL). WSFL can be used to describe a business process (that is, an execution flow from beginning to end), or a description of overall interactions between varying Web services with no specified sequence.

Looking at how these specifications work together, a Web service can be defined as a modular application that can be:

- Described using WSDL
- Published using UDDI
- Found using UDDI
- Bound using SOAP (or HTTP GET /POST)
- Invoked using SOAP (or HTTP GET/POST)
- Composed with other services into new services using WSFL

You can restrict access to Web services much as you would restrict access to Web sites that are not available to everyone. WebSphere® provides many options for controlling access and for authentication. The SOAP security extension, included with WebSphere Application Server 4.0, is intended to be a security architecture based on the SOAP Security specification, and on

widely accepted security technologies such as secure socket layer (SSL). When using HTTP as the transport mechanism, there are different ways to combine HTTP basic authentication, SSL, and SOAP signatures to handle varying needs of security and authentication.

Accessing Data

The sections that follow describe how to access DB2 data with Web services.

DB2 Data Access Through Web Services

IBM® is enabling its key programming models and application servers with Web developing tools to automatically generate Web services from existing artifacts and stored procedures. The sections that follow describe another way to submit SQL queries and, if you require, control the format of the returned Web service operations are planned to be supported:

- XML-based query or storage. In other words, an XML document is stored in DB2® realational tables and composed again on retrieval. This method of operation requires the presence of the DB2 XML Extender.
- SQL-based operations, such as calling stored procedures, or inserting, updating, and deleting DB2 data.

Related concepts:

- "DB2 Data Access Using XML-Based Queries" on page 311
- "DB2 Data Access Using SQL-Based Queries" on page 311

DB2 Data Access Using XML-Based Queries

XML-based query allows you to compose XML documents from relational data. You can also deconstruct an XML document into its component parts and store it in relational tables. Part of the underlying suport for this functionality is provided by DB2® XML Extender. The store and retrieve operations are handled by special stored procedures that are shipped with DB2 XML Extender.

One of the inputs into both storage and retrieval is the user-specified mapping file that creates the association between relational data and XML document structure. This mapping file is called a *document access defintion* (DAD), and provides a way that you can create an XML document with the XML elements and attributes named as you require and with the shape that you want. The focus of this approach is in moving and manipulating XML documents.

DB2 Data Access Using SQL-Based Queries

In Web services, SQL-based query is simply the ability to send SQL statements, including stored procedure calls, to DB2® and to return results

with the default tagging. The focus of this approach is to move the data in and out of the database, and not on shaping the results in a particular way.

SQL-based query does not require the use of DB2 XML Extender, because there is no user-defined mapping of SQL data to XML elements and attributes. Instead, the data is returned using only a simple mapping of SQL data types, using column names as elements.

However, if you are using DB2 XML Extender to store XML documents within a single column of a table, you can use SQL-based query to retrieve those documents intact as a character large object (CLOB), or to invoke the user-defined functions that extract parts of the document. Another feature of DB2 XML Extender is the ability to store frequently accessed data in side tables, thereby enabling fast searches on XML documents that are stored in columns.

Another useful thing you can do with SQL-based query is to invoke DB2 stored procedures. Stored procedures are natural for conversion to Web services since they are themselves an encapsulation of programming logic and database access. A Web service invocation of a stored procedure makes it possible to dynamically provide input parameters and to retrieve results.

Document Access Definition Extension File

Both the XML-based and SQL-based forms of querying are controlled by a configuration file called a *document access definition extension* (DADX). The DADX configuration file defines the operations that can be performed by the Web service. For example, you might have a DADX file that specifies the operations to find all orders for parts, find all orders for parts with a particular color, and orders for parts that are above a certain sepcified price. (The color or price can be specified at runtime as input parameters by using host-variable style notation in the query.)

After you create a DADX file, WebSphere Studio Application Developer can automatically generate the WSDL description of the interfaces, and publish the interfaces to a UDDI registry or some other service directory. WebSphere Studio Application Developer will also generate the artifacts needed to deploy the Web service to a WebSphere Application Server, and generate the client proxies, which you can use both for testing, and as a basis for building the client part of your Web application.

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 313

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 314
- "Enterprise Java Beans" on page 317

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 315
- "Enterprise Java Beans" on page 317

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:

- "Programming Considerations for Java" on page 257

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.

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 propogates 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 DB2 JDBC driver also implements part of JTA so that WebSphere and DB2 can provide coordinated distributed transactions.

Note: It is not necessary to configure DB2 to be JTA enabled in the WebSphere environment because the DB2 JDBC driver automatically detects this environment. Currently JTA support is only provided in the DB2 JDBC Type 2 driver.

DB2 JDBC driver provides two DataSource classes:
- `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`
- `COM.ibm.db2.jdbc.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`

Enterprise Java Beans

The Enterprise Java™ beans architecture is component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise Java beans architecture is scalable, transactional, and multi-user secure. These applications may 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 the business services and are not shared between users. Entity beans are multi-user, distributed transactional objects representing persistent data. The transactional boundaries of a EJB application can be set by specifying either container-managed or bean-managed transactions

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) via 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 DB2 database. The sample utilizes the Model-View-Controller (MVC) 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 that is implemented using Enterprise Java beans. The Model component of the sample consists of two EJBs, one session bean and one entity bean. The CMP entity bean, Employee, represent the distributed transactional objects that representing the persistent data in EMPLOYEE table of the sample database.

The purpose to use container-managed persistence (CMP) bean is to show how easy it is on development. 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). Because of this flexibility, even if you redeploy the same entity bean on different Java 2 Enterprise Edition servers that use different databases. 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 make more efficient in distributed transaction than that when the EJB client access the entity bean directly. The access to DB2 database can be provided from Session bean or Entity bean. The two services of the sample application demonstrate both approaches to access DB2 database according to the characteristics of the services. In the Service one, Entity bean is used:

```
//=================================================
// This method returns an employee's information by
// interactive 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)
{
......
```

The one line of code is needed to access DB2 database. In the service of displaying employee numbers, the Session bean, AccessEmployee, directly accesses 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;
```

The sample program `AccessEmployee.ear` uses Enterprise Java beans to implement a Java 2 Enterprise Edition application to access DB2 database. You can find this sample in the `SQLLIB/samples/websphere` directory.

Related reference:

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

WebSphere

The sections that follow describe WebSphere connection pooling and statement caching.

Connections to Enterprise Data

With companies relying more and more on their stored data there is a need to have it all on large systems such as the zSeries™ servers. With this new need for consolidation, Web applications need ways to get to the enterprise data.

DB2® Connect gives a Windows® or UNIX-based applications 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 concepts:

- "JDBC 2.1 Core API Restrictions by the DB2 JDBC Type 2 Driver" on page 272
- "JDBC 2.1 Optional Package API Support by the DB2 JDBC Type 2 Driver" on page 273

Related reference:

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

Parameters for Tuning WebSphere Connection Pools

Performance improvements can be made by correctly tuning the parameters on the connection pool. This section details each of the properties found on the Connection Pooling tab and how they can be tuned for optimal performance.

The following properties can be found on both the Connection Pooling tab of the data source creation window, and on the Connection Pooling tab of the data source properties window:

Minimum Pool Size
The minimum number of connections that the connection pool can hold. By default, this value is 1. Any non-negative integer is a valid value for this property. The minimum pool size can affect the performance of an application. Smaller pools require less overhead when the demand is low because fewer connections are being held open to the database. On the other hand, when the demand is high, the first applications will experience a slow response because new connections will have to be created if all others in the pool are in use.

Maximum Pool Size
The maximum number of connections the connection pool can hold. By default, this value is 10. Any positive integer is a valid value for

this property. The maximum pool size can affect the performance of an application. Larger pools require more overhead when demand is high because there are more connections open to the database at peak demand. These connections persist until they are idled out of the pool. On the other hand, if the maximum is smaller, there might be longer wait times or possible connection timeout errors during peak times. The database must be able to support the maximum number of connections in the application server in addition to any load that it may have outside of the application server.

Connection Timeout
The maximum number of seconds that an application waits for a connection from the pool before timing out and throwing a ConnectionWaitTimeoutException to the application. The default value is 180 seconds (3 minutes). Any non-negative integer is a valid value for this property. Setting this value to 0 disables the connection timeout.

Idle Timeout
The number of seconds that a connection can remain free in the pool before the connection is removed from the pool. The default value is 1800 seconds (30 minutes). Any non-negative integer is a valid value for this property. Connections need to be idled out of the pool because keeping connections open to the database can cause memory problems with the database. However, not all connections are idled out of the pool, even if they are older than the Idle Timeout setting. A connection is not idled if removing the connection would cause the pool to shrink below its minimum size. Setting this value to 0 disables the idle timeout.

Orphan Timeout
The number of seconds that an application is allowed to hold an inactive connection. The default is 1800 seconds (30 minutes). Any non-negative integer is a valid value for this property. If there has been no activity on an allocated connection for longer than the Orphan Timeout setting, the connection is marked for orphaning. After another Orphan Timeout number of seconds, if the connection still has had no activity, the connection is returned to the pool. If the application tries to use the connection again, it is thrown a StaleConnectionException. Connections that are enlisted in a transaction are not orphaned. Setting this value to 0 disables the orphan timeout.

Statement Cache Size
The number of cached prepared statements to keep for an entire connection pool. The default value is 100. Any non-negative integer is a valid value for this property. When a statement is cached, it helps performance, because a statement is retrieved from the cache if a

matching statement is found, instead of creating a new prepared statement (a more costly operation). The statement cache size does not change the programming model, only the performance of the application. The statement cache size is the number of cached statements for the entire pool, not for each connection. Setting the statement cache size in the administrative console is a new option in Version 4.0. In previous versions, this value could be set only by using a datasources.xml file. Use of datasources.xml has been deprecated in Version 4.0.

Disable Auto Connection Cleanup
Whether or not the connection is closed at the end of a transaction. The default is false, which indicates that when a transaction is completed, WebSphere® closes the connection and returns it to the pool. This means that any use of the connection after the transaction has ended results in a `StaleConnectionException`, because the connection has been closed and returned to the pool. This mechanism ensures that connections are not held indefinitely by the application. If the value is set to true, the connection is not returned to the pool at the end of a transaction. In this case, the application must return the connection to the pool by calling `close()`. If the application does not close the connection, the pool can run out of connections for other applications to use.

Each of these configurable parameters can determine how the resources are used in each pool. The most important two parameters are the Minimum Pool Size are the Maximum Pool Size. Below are some more in depth definitions and suggested uses of these two parameters:

Minimum Pool Size
The minimum number of connections that the connection pool can hold open to the database. The default value is 1. In versions 3.5 and 4.0, the pool does not create the minimum number of connections to the database up front. Instead, as additional connections are needed, new connections to the database are created, growing the pool. After the pool has grown to the minimum number of connections, it does not shrink below the minimum.

The correct minimum value for the pool can be determined by examining the applications that are using the pool. If it is determined, for example, that at least four connections are needed at any point in time, the minimum number of connections should be set to 4 to ensure that all requests can be fulfilled without connection wait timeout exceptions. At off-peak times, the pool shrinks back to this minimum number of connections. A good rule of thumb is to keep this number as small as possible to avoid holding connections unnecessarily open.

Maximum Pool Size

The maximum number of connections that the connection pool can hold open to the database. The pool holds this maximum number of connections open to the database at peak times. At off-peak times, the pool shrinks back to the minimum number of connections.

The best practice is to ensure that only one connection is required on a thread at any time. This avoids possible deadlocks when the pool is at maximum capacity and no connections are left to fulfill a connection request. Therefore, with one connection per thread, the maximum pool size can be set to the maximum number of threads.

When using servlets, this can be determined by looking at the `MaxConnections` property in the Servlet Engine. If multiple connections are required on a thread, the maximum pool size value can be determined using the following formula:

```
T *(C -1)+1
```

Where T is the number of threads, and C is the number of connections.

The other tuning parameters are very application specific when used for tuning. In each case it is very important to know roughly how long the average application will use a connection in the pool. For example, if all applications using a connection pool are know to only hold a connection for an average time of 5 seconds, with a maximum of 10 seconds, it may be useful to have an Orphan Timeout value of 10 or 15 seconds, but be prepared for the occasional `StaleConnectionException`.

Deliberately orphaning connections is not something that is recommended, however it may be useful it some problem determination scenarios where connections are being held for long periods of time when it is known that the application should only need a connection for a short period.

The Idle Timeout period is a very useful parameter if you are using a resource strapped machine. If you have set your maximum and minimum connection pool sizes properly, you may want to lower the Idle Timeout so that when usage of the pool is low, there are not connections sitting open that are not doing anything. Be very careful in how low to set this parameter though since setting it to low will introduce the cost of creating connections to more applications when a transformation from light load to heavy load begins.

Finally the Connection Timeout can be use to ensure that applications are not waiting around for ever for a connection. If all applications using a pool are know to only use connections for a few seconds, then it may be useful to reduce this parameter as well, however, don't the default for this parameter is quite low for heavily loaded systems to start with. For example, if most of the

applications using a pool are known to use a connection for at least 10 seconds (long running queries), then when this system is under load it is possible that the applications get backed up behind one another waiting for connections in the pool. The further back in line an application is the higher the chances are that it will get a Connection Timeout. Please note that this is a rare case, since not too many web applications have long running queries that a lot of users would be trying to access at the same time. Also note that if Minimum and Maximum connection pool size parameters are changed, you may no longer have the backup problem.

As in the last example it is important to consider all tuning parameters when tuning your system. Just changing the minimum and maximum pool size may give you a throughput and performance boost, but it may not help with resource contention if some of the other parameters are set improperly for the system.

As with most tuning, it is a good idea to try different settings and see what works best for your system

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 4. Other Programming Interfaces

Chapter 12. 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 330

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 13. 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 349

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 334

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 339
- "Host Variable References in REXX" on page 339
- "Indicator Variables in REXX" on page 339
- "LOB Host Variables in REXX" on page 341
- "LOB Host Variable Clearing in REXX" on page 344

Related reference:

- "Predefined REXX Variables" on page 339
- "Syntax for LOB Locator Declarations in REXX" on page 342
- "Syntax for LOB File Reference Declarations in REXX" on page 343
- "Supported SQL Data Types in REXX" on page 345

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.

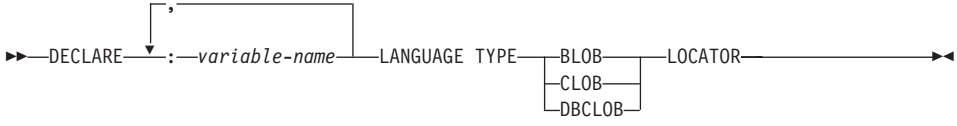
In REXX SQL, LOB types are determined from the string content of your host variable as follows:

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:

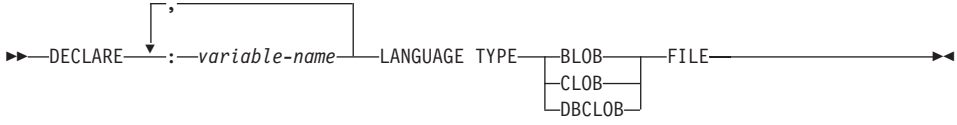
```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

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 345

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 18. 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
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

Table 18. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type[1]	REXX Data Type	SQL Column Type Description
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
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

Table 18. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type[1]	REXX Data Type	SQL Column Type Description

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 344

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 351

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 19. 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 352
- "Server Considerations for Calling Stored Procedures in REXX" on page 352
- "Retrieval of Precision and SCALE Values from SQLDA Decimal Fields" on page 353

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 345

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 14. Writing Applications Using the IBM OLE DB Provider for DB2 Servers

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 2.5 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 366

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
- 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 20. 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 20. 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	(NONE)	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 21. OLE DB Services

Enabled Services	DWORD Value
All services (default)	0xFFFFFFFF
All except pooling and AutoEnlistment	0xFFFFFFFE
All except client cursor	0xFFFFFFFB
All except pooling, enlistment and cursor	0xFFFFFFF0
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 and forward-only cursors, called *Server Cursors*. For updatable scrollable cursors, your application should use the OLE DB Cursor Service Component known as the Client Cursor. OLE DB native applications will have updatable and scrollable cursors available when the `IDataInitialize` or `IDBPromptInitialize` OLE DB core interface is used to connect to the database. This is because these interfaces automatically activate the OLE DB Cursor Service Component.

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 22. 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"

Table 22. 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
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"
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 23. 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											
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																						

Table 23. Data Conversions from OLE DB Types to DB2 Types (continued)

	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_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 364

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 24. 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

Table 24. Data Conversions from DB2 Types to OLE DB Types (continued)

	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_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																						
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 362

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.
- `ISQLErrorInfo` is not supported. The `IErrorLookUp`, `IErrorInfo`, and `IErrorRecords` interfaces are 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.
- Only a single set of parameters is supported. Multiple parameter sets are not yet supported.
- Named parameters are not supported by the IBM OLE DB Provider. When `ICommandWithParameters::MapParameterNames` is called, DB_S_ERRORSOCCURRED is always returned. Parameter names are ignored in `ICommandWithParameters::GetParameterInfo` and `ICommandWithParameters::SetParameterInfo`, since only ordinals are used.

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 25. 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

Table 25. 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
	ICommandPrepare	Yes	Yes
	ICommandProperties	Yes	Yes
	ICommandText	Yes	Yes
	ICommandWithParameters	Yes	Yes
	IColumnsInfo	Yes	Yes
	IColumnsRowset	No	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	No	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

Table 25. 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
	IErrorRecords	Yes	No
	ISQLErrorInfo (custom)	No	No
Multiple Results			
	IMultipleResults	Yes	Yes
	ISupportErrorInfo	Yes	Yes
RowSet			
	IAccessor	Yes	Yes
	IColumnsRowset	No	Yes
	IColumnsInfo	Yes	Yes
	IConvertType	Yes	Yes
	IChapteredRowset	No	No
	IConnectionPointContainer	No	Yes
	IDBAsynchStatus	No	No
	IParentRowset	No	No
	IRowset	Yes	Yes
	IRowsetChange	Cursor Service Component	Yes
	IRowsetChapterMember	No	No
	IRowsetFind	No	No
	IRowsetIdentity	Yes	Yes
	IRowsetIndex	No	No
	IRowsetInfo	Yes	Yes
	IRowsetLocate	Cursor Service Component	Yes
	IRowsetNotify (consumer)	No	No
	IRowsetRefresh	Cursor Service Component	Yes
	IRowsetResynch	Cursor Service Component	Yes
	IRowsetScroll	Cursor Service Component	Yes
	IRowsetUpdate	Cursor Service Component	Yes
	IRowsetView	No	No
	ISupportErrorInfo	Yes	Yes
Session			
	IAlterIndex	No	No

Table 25. 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
	IAlterTable	No	No
	IDBCreateCommand	Yes	Yes
	IDBSchemaRowset	Yes	Yes
	IGetDataSource	Yes	Yes
	IIndexDefinition	No	No
	IOpenRowset	Yes	Yes
	ISessionProperties	Yes	Yes
	ISupportErrorInfo	Yes	Yes
	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 26. 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
Data Source Information	DBPROPSET _DATASOURCEINFO	DBPROP_ACTIVESESSIONS	0	R
		DBPROP_ASYNCTXNABORT	VARIANT_FALSE	R
		DBPROP_ASYNCTXNCOMMIT	VARIANT_FALSE	R
		DBPROP_BYREFACCESSORS	VARIANT_FALSE	R

Table 26. Properties Supported by the IBM OLE DB Provider for DB2 (continued)

Property Group	Property Set	Properties	Default Value	R/W
		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
		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 Servers"	R
		DBPROP_PROVIDERNAME	"IBMDADB2.DLL"	R
		DBPROP_PROVIDEROLEDBVER	"02.00"	R
		DBPROP_PROVIDERVER	"08.01.0000"	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

Table 26. Properties Supported by the IBM OLE DB Provider for DB2 (continued)

Property Group	Property Set	Properties	Default Value	R/W
		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_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_CACHEDEFERRED	VARIANT_FALSE	R/W
		DBPROP_CANHOLDROWS	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
		DBPROP_IConvertType	VARIANT_TRUE	R
		DBPROP_IMultipleResults	VARIANT_TRUE	R
		DBPROP_IRowset	VARIANT_TRUE	R
		DBPROP_IRowChange	VARIANT_FALSE	R
		DBPROP_IRowsetFind	VARIANT_FALSE	R
		DBPROP_IRowsetIdentity	VARIANT_TRUE	R
		DBPROP_IRowsetInfo	VARIANT_TRUE	R
		DBPROP_IRowsetLocate	VARIANT_FALSE	R
		DBPROP_IRowsetScroll	VARIANT_FALSE	R
		DBPROP_IRowsetUpdate	VARIANT_FALSE	R
		DBPROP_ISequentialStream	VARIANT_TRUE	R
		DBPROP_ISupportErrorInfo	VARIANT_TRUE	R
		DBPROP_LITERALIDENTITY	VARIANT_TRUE	R
		DBPROP_LOCKMODE	DBPROPVAL_LM_SINGLEROW	R/W
		DBPROP_MAXOPENROWS	0	R/W
		DBPROP_MAXROWS	0	R/W
		DBPROP_QUICKRESTART	VARIANT_FALSE	R/W
		DBPROP_ROWTHREADMODEL	DBPROPVAL_RT_FREETHREAD	R
		DBPROP_SERVERCURSOR	VARIANT_TRUE	R
		DBPROP_UNIQUEROWS	VARIANT_FALSE	R
	DBPROPSET_DB2ROWSET	DBPROP_OPENROWSETSUPPORT	DBPROPVAL_ORS_TABLE	R

Table 26. Properties Supported by the IBM OLE DB Provider for DB2 (continued)

Property Group	Property Set	Properties	Default Value	R/W
		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 27. 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 372

Related tasks:

- "Building ADO Applications with Visual Basic" in the *Application Development Guide: Building and Running Applications*

Updatable Scrollable Cursors in ADO Applications

Because the IBM® OLE DB Provider for DB2® natively supports read-only and forward-only cursors, an ADO application that wants to access updatable scrollable cursors must set the cursor location to `adUseClient`.

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 yet supported.
- There is no support for default parameter values.
- For Visual Basic ADO applications, data controls are not supported for server side cursors. Data controls are available, however, for client side cursor applications.
- The `WithEvents` keyword cannot be used in the declaration of the recordset object for Visual Basic ADO applications using the read-only/forward-only server cursor; that is, when `Cursor Location` is specified as `adUseServer`.

IBM OLE DB Provider Support for ADO Methods and Properties

The IBM OLE DB Provider supports the following ADO methods and properties:

Table 28. 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
	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

Table 28. 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
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
	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) (Not Updatable Foward Only)	 Yes No Yes
	DefaultDataBase	IDBProperties DBPROP_CURRENTCATALOG	No
	IsolationLevel	ITransactionLocal DBPROP_SESS _AUTOCOMMITSOLEVELS	Yes
	Mode adModeRead adModeReadWrite adModeShareDenyNone adModeShareDenyRead adModeShareDenyWrite adModeShareExclusive adModeUnknown adModeWrite	IDBProperties DBPROP_INIT_MODE	 No Yes No No No No No No

Table 28. 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
	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 No
	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

Table 28. 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
Parameter Collection	Properties		Yes
RecordSet Methods	AddNew	IRowsetChange	Yes (Cursor Service)
	Cancel		Yes
	CancelBatch	IRowsetUpdate::Undo	Yes (Cursor Service)
	CancelUpdate		Yes (Cursor Service)
	Clone	IRowsetLocate	Yes (Cursor Service)
	Close	IAccessor IRowset	Yes
	CompareBookmarks		No
	Delete	IRowsetChange	Yes (Cursor Service)
	GetRows	IAccessor IRowset	Yes (Cursor Service)
	Move	IRowset IRowsetLocate	Server Cursor: forward only Cursor Service: scrollable
	MoveFirst	IRowset IRowsetLocate	Yes (Cursor Service)
	MoveNext	IRowset IRowsetLocate	Yes (Cursor Service)
	MoveLast	IRowsetLocate	Yes (Cursor Service)
	MovePrevious	IRowsetLocate	Yes (Cursor Service)
	NextRecordSet	IMultipleResults	Yes
	Open	ICommand IOpenRowset	Yes
	Requery	ICommand IOpenRowset	Yes
	Resync	IRowsetRefresh	Yes (Cursor Service)
	Supports	IRowsetInfo	Yes
	Update UpdateBatch	IRowsetChange IRowsetUpdate	Yes (Cursor Service) Yes (Cursor Service)
RecordSet Properties	AbsolutePage	IRowsetLocate IRowsetScroll	Yes (Cursor Service)
	AbsolutePosition	IRowsetLocate IRowsetScroll	Yes (Cursor Service)
	ActiveConnection	IDBCreateSession IDBInitialize	Yes
	BOF	(ADO specific)	

Table 28. 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
	Bookmark	IAccessor IRowsetLocate	Yes (Cursor Service)
	CacheSize	cRows in IRowsetLocate IRowset	Yes
	CursorType adOpenDynamic adOpenForwardOnly adOpenKeySet adOpenStatic	ICommandProperties	 No Yes No Yes (Cursor Service)
	EditMode	IRowsetUpdate	Yes (Cursor Service)
	EOF	(ADO specific)	
	Filter	IRowsetLocate IRowsetView IRowsetUpdate IViewChapter IViewFilter	No
	LockType	ICommandProperties	No
	MarshallOption		No
	MaxRecords	ICommandProperties IOpenRowset	Yes
	PageCount	IRowsetScroll	Yes (Cursor Service)
	PageSize	(ADO specific)	
	Sort	(ADO specific)	
	Source	(ADO specific)	
	State	(ADO specific)	
	Status	IRowsetUpdate	Yes (Cursor Service)
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, use one of the two OLE DB core interfaces, `IDBPromptInitialize` or `IDataInitialize`. Connecting to the data source in this way grants the application access to updatable scrollable cursors, instead of the read-only and forward-only cursors natively available. If IBMDADB2 is created directly by calling the COM API CoCreateInstance, the available cursors will be read-only and forward-only. 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 372

Related tasks:

- "Building ADO Applications with Visual C++" in the *Application Development Guide: Building and Running Applications*

Updatable Scrollable Cursors in ATL Applications and the IBM OLE DB Provider

If an ATL (Active Template Library) application requires updatable scrollable cursors, and its Data Access Consumer Object was generated by ATL COM AppWizard, the application must use the `OpenWithServiceComponents` method instead of the `Open` method call (which the wizard generates by default). If the `Open` method is used, the available cursors will be read-only and forward-only. The following example shows how to use the `OpenWithServiceComponents` method:

```
// The following line is generated by the wizard in the OpenDataSource method
//  hr = db.Open(_T("IBMDADB2"), &dbinit);
// Replace it with the following:
hr = db.OpenWithServiceComponents(_T("IBMDADB2"), &dbinit);
```

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, or Windows 2000

  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/`

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 IBMDABD2 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.

Part 5. General DB2 Application Concepts

Chapter 15. 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® 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.

Note: For double-byte and Unicode characters in GRAPHIC fields, the sort sequence is always IDENTITY.

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*
- "Character Comparisons Based on Collating Sequences" on page 385

Character Comparisons Based on Collating Sequences

Once a collating sequence is established, 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.

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 7. 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 8. 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, VARCHAR, LONG VARCHAR, and CLOB data in the Unicode database will be collated using the CESU-8 binary order instead of the UTF-8 binary order. CESU-8 is Compatibility Encoding Scheme for UTF-16: 8-Bit, and as of this writing, its specification is contained in the Draft Unicode Technical Report #26 available at the Unicode Technical 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, CHAR, VARCHAR, LONG VARCHAR, and CLOB data are stored in UTF-8, and GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB 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_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 163
- "Include Files for COBOL" on page 214
- "Include Files for FORTRAN" on page 239

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.

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.

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 176
- "Derivation of Code Page Values" on page 391
- "Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations" on page 404

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 29. 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'

Table 29. Code Points for Special Double-Byte Characters (continued)

Code Page	Double-Byte Percentage	Double-Byte Underscore	Double-Byte Space	Double-Byte Substitution Character
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 402

Remote Stored Procedures and UDFs

When coding stored procedures that will be running remotely, the following considerations apply:

- Data in a stored procedure must be in the database code page.
- Data passed to or from a stored procedure using an SQLDA with a character data type must really contain character data. Numeric data and data structures must never be passed with a character type if the client

application code page is different from the database code page. The reason for this is that the server will convert all character data in an SQLDA. 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 DB2® DARI stored procedures and UDFs, 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 stored procedure or UDF.

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 397

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 396

Character Conversion Between Different Code Pages

Ideally, for optimal performance, your applications should always use the same code page as your database. 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 database:

  This conversion will occur on the application client for both conversions from the application code page to the database code page and from the database code page to the application code page.

  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.

- 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 194

Related reference:

- "PRECOMPILE" 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 397

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 408

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 30. 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 31. 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 32. 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 33. 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 34. 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 400
- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 415

Related reference:

- "Supported territory codes and code pages" in the *Administration Guide: Planning*
- "Extended UNIX Code (EUC) Character Sets" on page 402

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

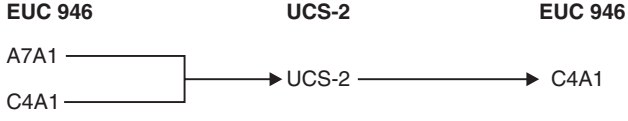
imposes fewer restrictions than administering a double-byte database from an EUC client. Double-byte character strings typically are equal or shorter than the corresponding EUC character string. This characteristic will generally lead to fewer problems caused by enforcing the character string length 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.

If you require the code page conversion tables for EUC code pages 946 (Traditional Chinese EUC) or 950 (Traditional Chinese Big-5) and UCS-2, see the online Product and Service Technical Library.

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 35. 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

Table 35. SQLCA.SQLERRD Settings on CONNECT (continued)

Application Code Page	Database Code Page	SQLERRD(1)	SQLERRD(2)
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 415

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 391
- "Code Page Conversion Expansion Factor" on page 400
- "Code Page Conversion String-Length Overflow in Mixed Code Set Environments" on page 415

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). For a Unicode database, GRAPHIC data is in UCS-2 big-endian order. However, when 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.

Note: The information that applies to applications in mixed code sets also applies to applications that connect to Unicode databases.

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.

For applications that connect to a Unicode database, GRAPHIC data is already in Unicode. For applications that connect to DBCS databases, GRAPHIC data is converted between the application DBCS code page and the database DBCS code page. Unicode applications should perform the necessary conversions to and from Unicode themselves, or should set the WCHARTYPE CONVERT option and use `wchar_t` for graphic data.

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 176

Chapter 16. 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 OS/390 and z/OS 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 36. 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 421

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" 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:

- "Sync Point Manager Log File Path configuration parameter - spm_log_path" in the *Administration Guide: Performance*
- "Transaction Resync Interval configuration parameter - resync_interval" in the *Administration Guide: Performance*
- "Transaction Manager Database Name configuration parameter - tm_database" in the *Administration Guide: Performance*
- "Transaction Processor Monitor Name configuration parameter - tp_mon_name" in the *Administration Guide: Performance*

- "Lock Timeout configuration parameter - locktimeout" in the *Administration Guide: Performance*
- "Sync Point Manager Name configuration parameter - spm_name" in the *Administration Guide: Performance*
- "Sync Point Manager Log File Size configuration parameter - spm_log_file_sz" in the *Administration Guide: Performance*
- "Sync Point Manager Resync Agent Limit configuration parameter - spm_max_resync" 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 SQLCODEs, 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 OS/390 and z/OS, OS/400, or DB2 for VSE & VM) that you intend to support.

Related concepts:

- "Applications in Host or iSeries Environments" on page 481

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 207

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 428

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 427

Related reference:

- "Lock Timeout configuration parameter - locktimeout" in the *Administration Guide: Performance*

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 CONSTRAINTS)
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 CONSTRAINTS),
         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 OS/390 and z/OS.

- 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 UDB" in the *Administration Guide: Planning*
- "Multisite Update with DB2 Connect" on page 492

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.

Chapter 17. 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 436
- "Local Bypass in Partitioned Database Environments" on page 437

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" 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
     - 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 73
- "Package Creation Using the BIND Command" on page 83
- "Considerations for Using Buffered Inserts" on page 440
- "Restrictions on Using Buffered Inserts" on page 443

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 437

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 450
- "Merged Multiple SQLCA Structures" on page 451
- "Partition That Returns the Error" on page 452

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 452

Related reference:

- "RESTART DATABASE" 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 SQLCODEs 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" in the *Command Reference*
- "UPDATE MONITOR SWITCHES" in the *Command Reference*
- "db2trc - Trace" in the *Command Reference*
- "db2support - Problem Analysis and Environment Collection Tool" in the *Command Reference*

Chapter 18. Common DB2 Application Techniques

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 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 or ALTER TABLE statement.

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)"

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 PREVAL 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 statitics 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)"

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; however, you cannot nest savepoints. 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. 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 improve the clarity of your code, you can choose a meaningful name for the savepoint. For example:

```
SAVEPOINT savepoint1 ON ROLLBACK RETAIN CURSORS
```

RELEASE SAVEPOINT
To release a savepoint, issue a RELEASE SAVEPOINT SQL statement. If you do not explicitly release a savepoint with a RELEASE SAVEPOINT SQL statement, it is released at the end of the transaction. For example:

```
RELEASE SAVEPOINT savepoint1
```

ROLLBACK TO SAVEPOINT
To rollback to a savepoint, issue a ROLLBACK TO SAVEPOINT SQL statement. For example:

```
ROLLBACK TO SAVEPOINT
```

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:

Atomic compound SQL
DB2 does not enable you to use savepoints within atomic compound SQL. You cannot use atomic compound SQL within a savepoint.

Nested Savepoints
DB2 does not support the use of a savepoint within another savepoint.

Triggers
DB2 does not support the use of savepoints in triggers.

SET INTEGRITY statement
Within a savepoint, DB2 treats SET INTEGRITY statements as DDL statements.

Related concepts:

• "Savepoints and Data Definition Language (DDL)" on page 469

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
: Within a savepoint, you can create a table with the NOT LOGGED INITIALLY property, or alter a table to have the NOT LOGGED INITIALLY property. For these savepoints, however, DB2 treats ROLLBACK TO SAVEPOINT statements as ROLLBACK WORK statements and rolls back the entire transaction.

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.

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 437

Related reference:

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

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 397
- "DB2 Stored Procedures" on page 22

Related tasks:
- "Specifying row blocking to reduce overhead" in the *Administration Guide: Performance*

Part 6. Appendixes

Appendix A. Supported SQL Statements

The following table:

- Lists all the supported SQL statements in DB2 Universal Database for Linux, UNIX, and Windows operating systems
- Indicates (with an 'X') if they can be executed dynamically
- Indicates (with an 'X') if they are supported by the command line processor (CLP)
- Indicates (with an 'X' or DB2 CLI function name) if the statement can be executed using the DB2 Call Level Interface (DB2 CLI)
- Indicates (with an 'X') if the statement can be executed in an SQL procedure

Table 37. SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
ALLOCATE CURSOR				X
assignment statement				X
ASSOCIATE LOCATORS				X
ALTER { BUFFERPOOL, NICKNAME,[10] NODEGROUP, SERVER,[10] TABLE, TABLESPACE, USER MAPPING,[10] TYPE, VIEW }	X	X	X	
BEGIN DECLARE SECTION[2]				
CALL		X[9]	X[4]	X
CASE statement				X
CLOSE		X	`SQLCloseCursor()`, `SQLFreeStmt()`	X
COMMENT ON	X	X	X	X
COMMIT	X	X	`SQLEndTran()`, `SQLTransact()`	X
Compound SQL (Embedded)			X[4]	
compound statement				X
CONNECT (Type 1)		X	`SQLBrowseConnect()`, `SQLConnect()`, `SQLDriverConnect()`	

Table 37. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
CONNECT (Type 2)		X	`SQLBrowseConnect()`, `SQLConnect()`, `SQLDriverConnect()`	
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING,[10] INDEX, INDEX EXTENSION, METHOD, NICKNAME,[10] NODEGROUP, PROCEDURE, SCHEMA, SERVER, TABLE, TABLESPACE, TRANSFORM, TYPE MAPPING,[10] TRIGGER, USER MAPPING,[10] TYPE, VIEW, WRAPPER[10] }	X	X	X	X[11]
DECLARE CURSOR[2]		X	`SQLAllocStmt()`	X
DECLARE GLOBAL TEMPORARY TABLE	X	X	X	X
DELETE	X	X	X	X
DESCRIBE[8]		X	`SQLColAttributes()`, `SQLDescribeCol()`, `SQLDescribParam()`[6]	
DISCONNECT		X	`SQLDisconnect()`	
DROP	X	X	X	X[11]
END DECLARE SECTION[2]				
EXECUTE			`SQLExecute()`	X
EXECUTE IMMEDIATE			`SQLExecDirect()`	X
EXPLAIN	X	X	X	X
FETCH		X	`SQLExtendedFetch()` , `SQLFetch()`, `SQLFetchScroll()`	X
FLUSH EVENT MONITOR	X	X	X	
FOR statement				X
FREE LOCATOR			X[4]	X
GET DIAGNOSTICS				X

Table 37. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
GOTO statement				X
GRANT	X	X	X	X
IF statement				X
INCLUDE[2]				
INSERT	X	X	X	X
ITERATE				X
LEAVE statement				X
LOCK TABLE	X	X	X	X
LOOP statement				X
OPEN		X	`SQLExecute()`, `SQLExecDirect()`	X
PREPARE			`SQLPrepare()`	X
REFRESH TABLE	X	X	X	
RELEASE		X		X
RELEASE SAVEPOINT	X	X	X	X
RENAME TABLE	X	X	X	
RENAME TABLESPACE	X	X	X	
REPEAT statement				X
RESIGNAL statement				X
RETURN statement				X
REVOKE	X	X	X	
ROLLBACK	X	X	`SQLEndTran()`, `SQLTransact()`	X
SAVEPOINT	X	X	X	X
select-statement	X	X	X	X
SELECT INTO				X
SET CONNECTION		X	`SQLSetConnection()`	
SET CURRENT DEFAULT TRANSFORM GROUP	X	X	X	X
SET CURRENT DEGREE	X	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, `SQLSetConnectAttr()`	X

Table 37. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, `SQLSetConnectAttr()`	X
SET CURRENT PACKAGESET				
SET CURRENT QUERY OPTIMIZATION	X	X	X	X
SET CURRENT REFRESH AGE	X	X	X	X
SET EVENT MONITOR STATE	X	X	X	X
SET INTEGRITY	X	X	X	
SET PASSTHRU[10]	X	X	X	X
SET PATH	X	X	X	X
SET SCHEMA	X	X	X	X
SET SERVER OPTION[10]	X	X	X	X
SET transition-variable[5]	X	X	X	X
SIGNAL statement				X
SIGNAL SQLSTATE[5]	X	X	X	
UPDATE	X	X	X	X
VALUES INTO				X
WHENEVER[2]				
WHILE statement				X

Table 37. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic[1]	Command Line Processor (CLP)	Call Level Interface[3] (CLI)	SQL Procedure

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. An X indicates that you can execute this statement using either `SQLExecDirect()` or `SQLPrepare()` and `SQLExecute()`. If there is an equivalent DB2 CLI function, the function name is listed.
4. Although this statement is not dynamic, with DB2 CLI you can specify this statement when calling either `SQLExecDirect()`, or `SQLPrepare()` and `SQLExecute()`.
5. You can only use this within CREATE TRIGGER statements.
6. You can only use the SQL DESCRIBE statement to describe output, whereas with DB2 CLI you can also describe input (using the `SQLDescribeParam()` function).
7. You can only use the SQL FETCH statement to fetch one row at a time in one direction, whereas with the DB2 CLI `SQLExtendedFetch()` and `SQLFetchScroll()` functions, you can fetch into arrays. Furthermore, you can fetch in any direction, and at any position in the result set.
8. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command.
9. When CALL is issued through the command line processor, only certain procedures and their respective parameters are supported.
10. Statement is supported only for federated database servers.
11. SQL procedures can only issue CREATE and DROP statements for indexes, tables, and views.

Related reference:

- "DESCRIBE statement" in the *SQL Reference, Volume 2*
- "CALL Statement to Install, Replace, and Remove JAR Files" in the *Application Development Guide: Programming Server Applications*

Appendix B. 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 OS/390 and z/OS 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 OS/390 and z/OS, 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 OS/390 and z/OS 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 SQLTYPEs, supported by OS/400, are converted to fixed (packed) decimal SQLTYPEs.

- 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 OS/390 and z/OS 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 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 OS/390 and z/OS

ARI DB2 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 terrritory 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 SQLCODEs 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 OS/390 and z/OS
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 OS/390 and z/OS.

DB2 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 OS/390 and z/OS 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 188

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" 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 38. Isolation Levels

DB2 Connect	DB2 Universal Database for OS/390 and z/OS	DB2 for VSE & VM	DB2® UDB for iSeries	DB2 Universal Database
RR	RR	RR	note 1	RR
RS	note 2	RR	COMMIT(*ALL)	RS

Table 38. Isolation Levels (continued)

DB2 Connect	DB2 Universal Database for OS/390 and z/OS	DB2 for VSE & VM	DB2® UDB for iSeries	DB2 Universal Database
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 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 OS/390 and z/OS in ASCII format. This permits faster exchange of data between DB2 Connect and DB2 Universal Database for OS/390 and z/OS, 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 OS/390 and z/OS
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 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 OS/390 and z/OS 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 SQLCODEs for similar errors. You can handle this problem in either of two ways:

- Use the SQLSTATE instead of the SQLCODE for a particular error.

  SQLSTATEs have approximately the same meaning across the database products, and the products produce SQLSTATEs that correspond to the SQLCODEs.
- Map the SQLCODEs from one system to another system.

  By default, DB2® Connect maps SQLCODEs 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 OS/390 and z/OS 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 OS/390 and z/OS 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 OS/390 and z/OS, DB2 UDB for iSeries or DB2 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 OS/390 and z/OS 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 for VSE & VM Version 7 or higher, DB2 Universal Database for OS/390 and z/OS 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 22

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 SQLSTATEs 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 UDB" 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 for VSE & VM extended dynamic SQL statements are rejected with -104 and syntax error SQLCODEs.

Appendix C. 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 9. 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	ǀ 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 10. 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	Đ 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	| 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 11. Code Page 850

Related concepts:

- "Collating Sequences" on page 383

Related reference:

- "sqlecrea - Create Database" in the *Administrative API Reference*

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

DB2 Universal Database technical information

Overview of DB2 Universal Database technical information

DB2 Universal Database technical information can be obtained in the following formats:

- Books (PDF and hard-copy formats)
- A topic tree (HTML format)
- Help for DB2 tools (HTML format)
- Sample programs (HTML format)
- Command line help
- Tutorials

This section is an overview of the technical information that is provided and how you can access it.

Categories of DB2 technical information

The DB2 technical information is categorized by the following headings:

- Core DB2 information
- Administration information
- Application development information
- Business intelligence information
- DB2 Connect information
- Getting started information
- Tutorial information
- Optional component information
- Release notes

The following tables describe, for each book in the DB2 library, the information needed to order the hard copy, print or view the PDF, or locate the HTML directory for that book. A full description of each of the books in the DB2 library is available from the IBM Publications Center at www.ibm.com/shop/publications/order

The installation directory for the HTML documentation CD differs for each category of information:

htmlcdpath/doc/htmlcd/%L/*category*

where:

- *htmlcdpath* is the directory where the HTML CD is installed.
- *%L* is the language identifier. For example, en_US.
- *category* is the category identifier. For example, core for the core DB2 information.

In the PDF file name column in the following tables, the character in the sixth position of the file name indicates the language version of a book. For example, the file name `db2d1e80` identifies the English version of the *Administration Guide: Planning* and the file name `db2d1g80` identifies the German version of the same book. The following letters are used in the sixth position of the file name to indicate the language version:

Language	**Identifier**
Arabic	w
Brazilian Portuguese	b
Bulgarian	u
Croatian	9
Czech	x
Danish	d
Dutch	q
English	e
Finnish	y
French	f
German	g
Greek	a
Hungarian	h
Italian	i
Japanese	j
Korean	k
Norwegian	n
Polish	p
Portuguese	v
Romanian	8
Russian	r
Simp. Chinese	c
Slovakian	7
Slovenian	l
Spanish	z
Swedish	s
Trad. Chinese	t
Turkish	m

No form number indicates that the book is only available online and does not have a printed version.

Core DB2 information

The information in this category cover DB2 topics that are fundamental to all DB2 users. You will find the information in this category useful whether you are a programmer, a database administrator, or you work with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

The installation directory for this category is doc/htmlcd/%L/core.

Table 39. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x80
IBM DB2 Universal Database Glossary	No form number	db2t0x80
IBM DB2 Universal Database Master Index	SC09-4839	db2w0x80
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840	db2m1x80
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841	db2m2x80
IBM DB2 Universal Database What's New	SC09-4848	db2q0x80

Administration information

The information in this category covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

The installation directory for this category is doc/htmlcd/%L/admin.

Table 40. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x80
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x80
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x80
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x80

Table 40. Administration information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx80
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax80
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx80
IBM DB2 Universal Database Federated Systems Guide	GC27-1224	db2fpx80
IBM DB2 Universal Database Guide to GUI Tools for Administration and Development	SC09-4851	db2atx80
IBM DB2 Universal Database Replication Guide and Reference	SC27-1121	db2e0x80
IBM DB2 Installing and Administering a Satellite Environment	GC09-4823	db2dsx80
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x80
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x80
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x80

Application development information

The information in this category is of special interest to application developers or programmers working with DB2. You will find information about supported languages and compilers, as well as the documentation required to access DB2 using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLj, and CLI. If you view this information online in HTML you can also access a set of DB2 sample programs in HTML.

The installation directory for this category is doc/htmlcd/%L/ad.

Table 41. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx80
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x80
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x80
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx80
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx80

Business intelligence information

The information in this category describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

The installation directory for this category is doc/htmlcd/%L/wareh.

Table 42. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Information Catalog Center Administration Guide	SC27-1125	db2dix80
IBM DB2 Warehouse Manager Installation Guide	GC27-1122	db2idx80

DB2 Connect information

The information in this category describes how to access host or iSeries data using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

The installation directory for this category is doc/htmlcd/%L/conn.

Table 43. DB2 Connect information

Name	Form number	PDF file name
APPC, CPI-C, and SNA Sense Codes	No form number	db2apx80
IBM Connectivity Supplement	No form number	db2h1x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x80
IBM DB2 Connect User's Guide	SC09-4835	db2c0x80

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

The installation directory for this category is doc/htmlcd/%L/start.

Table 44. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832	db2itx80
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx80
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x80
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837	db2iyx80
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x80

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

The installation directory for this category is doc/htmlcd/%L/tutr.

Table 45. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux80
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax80
Development Center Tutorial for Video Online using Microsoft Visual Basic	No form number	db2tdx80
Information Catalog Center Tutorial	No form number	db2aix80
Video Central for e-business Tutorial	No form number	db2twx80
Visual Explain Tutorial	No form number	db2tvx80

Optional component information

The information in this category describes how to work with optional DB2 components.

The installation directory for this category is doc/htmlcd/%L/opt.

Table 46. Optional component information

Name	Form number	PDF file name
IBM DB2 Life Sciences Data Connect Planning, Installation, and Configuration Guide	GC27-1235	db2lsx80
IBM DB2 Spatial Extender User's Guide and Reference	SC27-1226	db2sbx80
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x80

Table 46. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Net Search Extender Administration and Programming Guide **Note:** HTML for this document is not installed from the HTML documentation CD.	SH12-6740	N/A

Release notes

The release notes provide additional information specific to your product's release and FixPak level. They also provides summaries of the documentation updates incorporated in each release and FixPak.

Table 47. Release notes

Name	Form number	PDF file name	HTML directory
DB2 Release Notes	See note.	See note.	doc/prodcd/%L/db2ir where %L is the language identifier.
DB2 Connect Release Notes	See note.	See note.	doc/prodcd/%L/db2cr where %L is the language identifier.
DB2 Installation Notes	Available on product CD-ROM only.	Available on product CD-ROM only.	

Note: The HTML version of the release notes is available from the Information Center and on the product CD-ROMs. To view the ASCII file:

- On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L represents the locale name and DB2DIR represents:
  - /usr/opt/db2_08_01 on AIX
  - /opt/IBM/db2/V8.1 on all other UNIX operating systems
- On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.

Related tasks:

- "Printing DB2 books from PDF files" on page 529

- "Ordering printed DB2 books" on page 530
- "Accessing online help" on page 530
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 534
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 535

Printing DB2 books from PDF files

You can print DB2 books from the PDF files on the *DB2 PDF Documentation* CD. Using Adobe Acrobat Reader, you can print either the entire book or a specific range of pages.

Prerequisites:

Ensure that you have Adobe Acrobat Reader. It is available from the Adobe Web site at www.adobe.com

Procedure:

To print a DB2 book from a PDF file:

1. Insert the *DB2 PDF Documentation* CD. On UNIX operating systems, mount the DB2 PDF Documentation CD. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start Adobe Acrobat Reader.
3. Open the PDF file from one of the following locations:
   - On Windows operating systems:

     x:\doc*language* directory, where *x* represents the CD-ROM drive letter and *language* represents the two-character territory code that represents your language (for example, EN for English).
   - On UNIX operating systems:

     /cdrom/doc/*%L* directory on the CD-ROM, where */cdrom* represents the mount point of the CD-ROM and *%L* represents the name of the desired locale.

Related tasks:

- "Ordering printed DB2 books" on page 530
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 534
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 535

Related reference:

- "Overview of DB2 Universal Database technical information" on page 521

Ordering printed DB2 books

Procedure:

To order printed books:

- Contact your IBM authorized dealer or marketing representative. To find a local IBM representative, check the IBM Worldwide Directory of Contacts at www.ibm.com/shop/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.
- Visit the IBM Publications Center at www.ibm.com/shop/publications/order

Related tasks:

- "Printing DB2 books from PDF files" on page 529
- "Finding topics by accessing the DB2 Information Center from a browser" on page 532
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 535

Related reference:

- "Overview of DB2 Universal Database technical information" on page 521

Accessing online help

The online help that comes with all DB2 components is available in three types:

- Window and notebook help
- Command line help
- SQL statement help

Window and notebook help explain the tasks that you can perform in a window or notebook and describe the controls. This help has two types:

- Help accessible from the **Help** button
- Infopops

The **Help** button gives you access to overview and prerequisite information. The infopops describe the controls in the window or notebook. Window and notebook help are available from DB2 centers and components that have user interfaces.

Command line help includes Command help and Message help. Command help explains the syntax of commands in the command line processor. Message help describes the cause of an error message and describes any action you should take in response to the error.

SQL statement help includes SQL help and SQLSTATE help. DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the syntax of SQL statements (SQL states and class codes).

Note: SQL help is not available for UNIX operating systems.

Procedure:

To access online help:

- For window and notebook help, click **Help** or click that control, then click **F1**. If the **Automatically display infopops** check box on the **General** page of the **Tool Settings** notebook is selected, you can also see the infopop for a particular control by holding the mouse cursor over the control.
- For command line help, open the command line processor and enter:
  - For Command help:

    ```
    ? command
    ```

    where *command* represents a keyword or the entire command.

    For example, `? catalog` displays help for all the CATALOG commands, while `? catalog database` displays help for the CATALOG DATABASE command.
  - For Message help:

    ```
    ? XXXnnnnn
    ```

    where *XXXnnnnn* represents a valid message identifier.

    For example, `? SQL30081` displays help about the SQL30081 message.
- For SQL statement help, open the command line processor and enter:
  - For SQL help:

    ```
    ? sqlstate or ? class code
    ```

    where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

    For example, `? 08003` displays help for the 08003 SQL state, while `? 08` displays help for the 08 class code.
  - For SQLSTATE help:

```
help statement
```

where *statement* represents an SQL statement.

For example, help SELECT displays help about the SELECT statement.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 532
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 535

Finding topics by accessing the DB2 Information Center from a browser

The DB2 Information Center accessed from a browser enables you to access the information you need to take full advantage of DB2 Universal Database and DB2 Connect. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, metadata, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser is composed of the following major elements:

Navigation tree
The navigation tree is located in the left frame of the browser window. The tree expands and collapses to show and hide topics, the glossary, and the master index in the DB2 Information Center.

Navigation toolbar
The navigation toolbar is located in the top right frame of the browser window. The navigation toolbar contains buttons that enable you to search the DB2 Information Center, hide the navigation tree, and find the currently displayed topic in the navigation tree.

Content frame
The content frame is located in the bottom right frame of the browser window. The content frame displays topics from the DB2 Information Center when you click on a link in the navigation tree, click on a search result, or follow a link from another topic or from the master index.

Prerequisites:

To access the DB2 Information Center from a browser, you must use one of the following browsers:

- Microsoft Explorer, version 5 or later
- Netscape Navigator, version 6.1 or later

Restrictions:

The DB2 Information Center contains only those sets of topics that you chose to install from the *DB2 HTML Documentation CD*. If your Web browser returns a `File not found` error when you try to follow a link to a topic, you must install one or more additional sets of topics *DB2 HTML Documentation CD*.

Procedure:

To find a topic by searching with keywords:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

To find a topic in the navigation tree:

1. In the navigation tree, click the book icon of the category of topics related to your area of interest. A list of subcategories displays underneath the icon.
2. Continue to click the book icons until you find the category containing the topics in which you are interested. Categories that link to topics display the category title as an underscored link when you move the cursor over the category title. The navigation tree identifies topics with a page icon.
3. Click the topic link. The topic displays in the content frame.

To find a topic or term in the master index:

1. In the navigation tree, click the “Index” category. The category expands to display a list of links arranged in alphabetical order in the navigation tree.
2. In the navigation tree, click the link corresponding to the first character of the term relating to the topic in which you are interested. A list of terms with that initial character displays in the content frame. Terms that have multiple index entries are identified by a book icon.
3. Click the book icon corresponding to the term in which you are interested. A list of subterms and topics displays below the term you clicked. Topics are identified by page icons with an underscored title.
4. Click on the title of the topic that meets your needs. The topic displays in the content frame.

Related concepts:

- "Accessibility" on page 541
- "DB2 Information Center for topics" on page 543

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 534
- "Updating the HTML documentation installed on your machine" on page 536
- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 538
- "Searching the DB2 documentation" on page 539

Related reference:

- "Overview of DB2 Universal Database technical information" on page 521

Finding product information by accessing the DB2 Information Center from the administration tools

The DB2 Information Center provides quick access to DB2 product information and is available on all operating systems for which the DB2 administration tools are available.

The DB2 Information Center accessed from the tools provides six types of information.

Tasks Key tasks you can perform using DB2.

Concepts
Key concepts for DB2.

Reference
DB2 reference information, such as keywords, commands, and APIs.

Troubleshooting
Error messages and information to help you with common DB2 problems.

Samples
Links to HTML listings of the sample programs provided with DB2.

Tutorials
Instructional aid designed to help you learn a DB2 feature.

Prerequisites:

Some links in the DB2 Information Center point to Web sites on the Internet. To display the content for these links, you will first have to connect to the Internet.

Procedure:

To find product information by accessing the DB2 Information Center from the tools:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter `db2ic`.
2. Click the tab of the information type related to the information you are attempting to find.
3. Navigate through the tree and click on the topic in which you are interested. The Information Center will then launch a Web browser to display the information.
4. To find information without browsing the lists, click the **Search** icon to the right of the list.

   Once the Information Center has launched a browser to display the information, you can perform a full-text search by clicking the **Search** icon in the navigation toolbar.

Related concepts:

- "Accessibility" on page 541
- "DB2 Information Center for topics" on page 543

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 532
- "Searching the DB2 documentation" on page 539

Viewing technical documentation online directly from the DB2 HTML Documentation CD

All of the HTML topics that you can install from the *DB2 HTML Documentation CD* can also be read directly from the CD. Therefore, you can view the documentation without having to install it.

Restrictions:

Because the following items are installed from the DB2 product CD and not the *DB2 HTML Documentation CD*, you must install the DB2 product to view these items:

- Tools help
- DB2 Quick Tour
- Release notes

Procedure:

1. Insert the *DB2 HTML Documentation* CD. On UNIX operating systems, mount the *DB2 HTML Documentation CD*. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start your HTML browser and open the appropriate file:
   - For Windows operating systems:

     ```
     e:\Program Files\sqllib\doc\htmlcd\%L\index.htm
     ```

     where *e* represents the CD-ROM drive, and %L is the locale of the documentation that you wish to use, for example, `en_US` for English.
   - For UNIX operating systems:

     ```
     /cdrom/Program Files/sqllib/doc/htmlcd/%L/index.htm
     ```

     where */cdrom/* represents where the CD is mounted, and %L is the locale of the documentation that you wish to use, for example, `en_US` for English.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 532
- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 538

Related reference:

- "Overview of DB2 Universal Database technical information" on page 521

Updating the HTML documentation installed on your machine

It is now possible to update the HTML installed from the *DB2 HTML Documentation CD* when updates are made available from IBM. This can be done in one of two ways:

- Using the Information Center (if you have the DB2 administration GUI tools installed).
- By downloading and applying a DB2 HTML documentation FixPak .

Note: This will NOT update the DB2 code; it will only update the HTML documentation installed from the *DB2 HTML Documentation CD*.

Procedure:

To use the Information Center to update your local documentation:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter `db2ic`.
2. Ensure your machine has access to the external Internet; the updater will download the latest documentation FixPak from the IBM server if required.
3. Select **Information Center —> Update Local Documentation** from the menu to start the update.
4. Supply your proxy information (if required) to connect to the external Internet.

The Information Center will download and apply the latest documentation FixPak, if one is available.

To manually download and apply the documentation FixPak :

1. Ensure your machine is connected to the Internet.
2. Open the DB2 support page in your Web browser at: www.ibm.com/software/data/db2/udb/winos2unix/support
3. Follow the link for version 8 and look for the ″Documentation FixPaks″ link.
4. Determine if the version of your local documentation is out of date by comparing the documentation FixPak level to the documentation level you have installed. This current documentation on your machine is at the following level: **DB2 v8.1 GA**.
5. If there is a more recent version of the documentation available then download the FixPak applicable to your operating system. There is one FixPak for all Windows platforms, and one FixPak for all UNIX platforms.
6. Apply the FixPak:
   - For Windows operating systems: The documentation FixPak is a self extracting zip file. Place the downloaded documentation FixPak in an empty directory, and run it. It will create a `setup` command which you can run to install the documentation FixPak.
   - For UNIX operating systems: The documentation FixPak is a compressed tar.Z file. Uncompress and untar the file. It will create a directory named `delta_install` with a script called `installdocfix`. Run this script to install the documentation FixPak.

Related tasks:

- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 538

Related reference:

- "Overview of DB2 Universal Database technical information" on page 521

Copying files from the DB2 HTML Documentation CD to a Web Server

The entire DB2 information library is delivered to you on the *DB2 HTML Documentation CD*, so you can install the library on a Web server for easier access. Simply copy to your Web server the documentation for the languages that you want.

Procedure:

To copy files from the *DB2 HTML Documentation CD* to a Web server, use the appropriate path:

- For Windows operating systems:

  ```
  E:\Program Files\sqllib\doc\htmlcd\%L\*.*
  ```

  where *E* represents the CD-ROM drive and *%L* represents the language identifier.
- For UNIX operating systems:

  ```
  /cdrom:Program Files/sqllib/doc/htmlcd/%L/*.*
  ```

  where *cdrom* represents the CD-ROM drive and *%L* represents the language identifier.

Related tasks:

- "Searching the DB2 documentation" on page 539

Related reference:

- "Supported DB2 interface languages, locales, and code pages" in the *Quick Beginnings for DB2 Servers*
- "Overview of DB2 Universal Database technical information" on page 521

Troubleshooting DB2 documentation search with Netscape 4.x

Most search problems are related to the Java support provided by web browsers. This task describes possible workarounds.

Procedure:

A common problem with Netscape 4.x involves a missing or misplaced security class. Try the following workaround, especially if you see the following line in the browser Java console:

```
Cannot find class  java/security/InvalidParameterException
```

- On Windows operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied *x*:Program Files\sqllib\doc\htmlcd*locale*\InvalidParameterException.class file to the java\classes\java\security\ directory relative to your Netscape browser installation, where *x* represents the CD-ROM drive letter and *locale* represents the name of the desired locale.

  Note: You may have to create the java\security\ subdirectory structure.

- On UNIX operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied */cdrom*/Program Files/sqllib/doc/htmlcd/*locale*/InvalidParameterException.class file to the java/classes/java/security/ directory relative to your Netscape browser installation, where *cdrom* represents the mount point of the CD-ROM and *locale* represents the name of the desired locale.

  Note: You may have to create the java/security/ subdirectory structure.

If your Netscape browser still fails to display the search input window, try the following:

- Stop all instances of Netscape browsers to ensure that there is no Netscape code running on the machine. Then open a new instance of the Netscape browser and try to start the search again.
- Purge the browser's cache.
- Try a different version of Netscape, or a different browser.

Related tasks:

- "Searching the DB2 documentation" on page 539

Searching the DB2 documentation

To search DB2's documentation, you need Netscape 6.1 or higher, or Microsoft's Internet Explorer 5 or higher. Ensure that your browser's Java support is enabled.

A pop-up search window opens when you click the search icon in the navigation toolbar of the Information Center accessed from a browser. If you are using the search for the first time it may take a minute or so to load into the search window.

Restrictions:

The following restrictions apply when you use the documentation search:

- Boolean searches are not supported. The boolean search qualifiers *and* and *or* will be ignored in a search. For example, the following searches would produce the same results:
  - servlets *and* beans
  - servlets *or* beans
- Wildcard searches are not supported. A search on *java** will only look for the literal string *java** and would not, for example, find *javadoc*.

In general, you will get better search results if you search for phrases instead of single words.

Procedure:

To search the DB2 documentation:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

Note: When you perform a search, the first result is automatically loaded into your browser frame. To view the contents of other search results, click on the result in results lists.

Related tasks:

- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 538

Online DB2 troubleshooting information

With the release of DB2® UDB Version 8, there will no longer be a *Troubleshooting Guide*. The troubleshooting information once contained in this guide has been integrated into the DB2 publications. By doing this, we are able to deliver the most up-to-date information possible. To find information on the troubleshooting utilities and functions of DB2, access the DB2 Information Center from any of the tools.

Refer to the DB2 Online Support site if you are experiencing problems and want help finding possible causes and solutions. The support site contains a

large, constantly updated database of DB2 publications, TechNotes, APAR (product problem) records, FixPaks, and other resources. You can use the support site to search through this knowledge base and find possible solutions to your problems.

Access the Online Support site at www.ibm.com/software/data/db2/udb/winos2unix/support, or by clicking the **Online Support** button in the DB2 Information Center. Frequently changing information, such as the listing of internal DB2 error codes, is now also available from this site.

Related concepts:

- "DB2 Information Center for topics" on page 543

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 534

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features in DB2® Universal Database Version 8:

- DB2 allows you to operate all features using the keyboard instead of the mouse. See "Keyboard Input and Navigation".
- DB2 enables you customize the size and color of your fonts. See "Accessible Display" on page 542.
- DB2 allows you to receive either visual or audio alert cues. See "Alternative Alert Cues" on page 542.
- DB2 supports accessibility applications that use the Java™ Accessibility API. See "Compatibility with Assistive Technologies" on page 542.
- DB2 comes with documentation that is provided in an accessible format. See "Accessible Documentation" on page 542.

Keyboard Input and Navigation

Keyboard Input

You can operate the DB2 Tools using only the keyboard. You can use keys or key combinations to perform most operations that can also be done using a mouse.

Keyboard Focus

In UNIX-based systems, the position of the keyboard focus is highlighted, indicating which area of the window is active and where your keystrokes will have an effect.

Accessible Display

The DB2 Tools have features that enhance the user interface and improve accessibility for users with low vision. These accessibility enhancements include support for customizable font properties.

Font Settings

The DB2 Tools allow you to select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

Non-dependence on Color

You do not need to distinguish between colors in order to use any of the functions in this product.

Alternative Alert Cues

You can specify whether you want to receive alerts through audio or visual cues, using the Tools Settings notebook.

Compatibility with Assistive Technologies

The DB2 Tools interface supports the Java Accessibility API enabling use by screen readers and other assistive technologies used by people with disabilities.

Accessible Documentation

Documentation for the DB2 family of products is available in HTML format. This allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

DB2 tutorials

The DB2® tutorials help you learn about various aspects of DB2 Universal Database. The tutorials provide lessons with step-by-step instructions in the areas of developing applications, tuning SQL query performance, working with data warehouses, managing metadata, and developing Web services using DB2.

Before you begin:

Before you can access these tutorials using the links below, you must install the tutorials from the *DB2 HTML Documentation* CD-ROM.

If you do not want to install the tutorials, you can view the HTML versions of the tutorials directly from the *DB2 HTML Documentation CD*. PDF versions of these tutorials are also available on the *DB2 PDF Documentation CD*.

Some tutorial lessons use sample data or code. See each individual tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

If you installed the tutorials from the *DB2 HTML Documentation* CD-ROM, you can click on a tutorial title in the following list to view that tutorial.

Business Intelligence Tutorial: Introduction to the Data Warehouse Center
Perform introductory data warehousing tasks using the Data Warehouse Center.

Business Intelligence Tutorial: Extended Lessons in Data Warehousing
Perform advanced data warehousing tasks using the Data Warehouse Center. (Not provided on CD. You can download this tutorial from the Downloads section of the Business Intelligence Solutions Web site at http://www.ibm.com/software/data/bi/.)

Development Center Tutorial for Video Online using Microsoft® Visual Basic
Build various components of an application using the Development Center Add-in for Microsoft Visual Basic.

Information Catalog Center Tutorial
Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Video Central for e-business Tutorial
Develop and deploy an advanced DB2 Web Services application using WebSphere® products.

Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 Information Center for topics

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 Universal Database™ and DB2 Connect™ in your business. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, the Information Catalog Center, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser has the following features:

Regularly updated documentation
Keep your topics up-to-date by downloading updated HTML.

Search
Search all of the topics installed on your workstation by clicking **Search** in the navigation toolbar.

Integrated navigation tree
Locate any topic in the DB2 library from a single navigation tree. The navigation tree is organized by information type as follows:

- Tasks provide step-by-step instructions on how to complete a goal.
- Concepts provide an overview of a subject.
- Reference topics provide detailed information about a subject, including statement and command syntax, message help, requirements.

Master index
Access the information in topics and tools help from one master index. The index is organized in alphabetical order by index term.

Master glossary
The master glossary defines terms used in the DB2 Information Center. The glossary is organized in alphabetical order by glossary term.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 532
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 534
- "Updating the HTML documentation installed on your machine" on page 536

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 which 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 information 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 illustrates 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.

ACF/VTAM
AISPO
AIX
AIX/6000
AIXwindows
AnyNet
APPN
AS/400
BookManager
CICS
C Set++
C/370
DATABASE 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eNetwork
eServer
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
iSeries
LAN Distance
MVS
MVS/ESA
MVS/XA
Net.Data
OS/390
OS/400
PowerPC
pSeries
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/DS
SQL/400
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WIN-OS/2
z/OS
zSeries

The following terms are trademarks or registered trademarks of other companies:

Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

Java and all Java-based trademarks and logos, and Solaris are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries or both and is licensed exclusively through X/Open Company Limited.

Other company, product, or service names may be trademarks or service marks of others.

Part Number: CT17TNA

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

(1P) P/N: CT17TNA

SC09-4826-00